Community Experience Distilled

Python Data Science Essentials

Become an efficient data science practitioner by thoroughly understanding the key concepts of Python

Alberto Boschetti
Luca Massaron

[PACKT] open source*
community experience distilled
PUBLISHING

Python Data Science Essentials

Become an efficient data science practitioner by thoroughly understanding the key concepts of Python

Alberto Boschetti

Luca Massaron

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Python Data Science Essentials

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

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

First published: April 2015

Production reference: 1240415

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

ISBN 978-1-78528-042-9

www.packtpub.com

Credits

Authors
Alberto Boschetti
Luca Massaron

Reviewers
Robert Dempsey
Daniel Frimer
Kevin Markham
Alberto Gonzalez Paje
Bastiaan Sjardin
Michele Usuelli
Zacharias Voulgaris, PhD

Commissioning Editor
Julian Ursell

Acquisition Editor
Subho Gupta

Content Development Editor
Merwyn D'souza

Technical Editor
Namrata Patil

Copy Editor
Vedangi Narvekar

Project Coordinator
Neha Bhatnagar

Proofreaders
Simran Bhogal
Faye Coulman
Safis Editing
Dan McMahon

Indexer
Priya Sane

Production Coordinator
Komal Ramchandani

Cover Work
Komal Ramchandani

About the Authors

Alberto Boschetti is a data scientist with expertise in signal processing and statistics. He holds a PhD in telecommunication engineering and currently lives and works in London. In his work projects, he faces challenges involving natural language processing (NLP), machine learning, and probabilistic graph models everyday. He is very passionate about his job and he always tries to stay updated on the latest developments in data science technologies by attending meetups, conferences, and other events.

I would like to thank my family, my friends, and my colleagues. Also, a big thanks to the open source community.

Luca Massaron is a data scientist and marketing research director who specializes in multivariate statistical analysis, machine learning, and customer insight, with over a decade of experience in solving real-world problems and generating value for stakeholders by applying reasoning, statistics, data mining, and algorithms. From being a pioneer of web audience analysis in Italy to achieving the rank of a top 10 Kaggler, he has always been passionate about everything regarding data and analysis and about demonstrating the potentiality of data-driven knowledge discovery to both experts and nonexperts. Favoring simplicity over unnecessary sophistication, he believes that a lot can be achieved in data science by understanding its essentials.

To Yukiko and Amelia, for their loving patience. "Roads go ever ever on, under cloud and under star, yet feet that wandering have gone turn at last to home afar".

About the Reviewers

Robert Dempsey is an experienced leader and technology professional specializing in delivering solutions and products to solve tough business challenges. His experience in forming and leading agile teams, combined with more than 14 years of experience in the field of technology, enables him to solve complex problems while always keeping the bottom line in mind.

Robert has founded and built three start-ups in technology and marketing, developed and sold two online applications, consulted Fortune 500 and Inc. 500 companies, and spoken nationally and internationally on software development and agile project management.

He is currently the head of data operations at ARPC, an econometrics firm based in Washington, DC. In addition, he's the founder of Data Wranglers DC, a group dedicated to improving the craft of data wrangling, as well as a board member of Data Community DC.

In addition to spending time with his growing family, Robert geeks out on Raspberry Pis and Arduinos and automates most of his life with the help of hardware and software.

Daniel Frimer has been an advocate for the Python language for 2 years now. With a degree in applied and computational math sciences from the University of Washington, he has spearheaded various automation projects in the Python language involving natural language processing, data munging, and web scraping. In his side projects, he has dived into a deep analysis of NFL and NBA player statistics for his fantasy sports teams.

Daniel has recently started working in SaaS at a private company for online health insurance shopping called Array Health, in support of day-to-day data analysis and the perfection of the integration between consumers, employers, and insurers. He has also worked with data-centric teams at Amazon, Starbucks, and Atlas International.

Kevin Markham is a computer engineer, a data science instructor for General Assembly in Washington, DC, and the cofounder of Causetown, an online cause marketing platform for small businesses. He is passionate about teaching data science and machine learning and enjoys both Python and R. He founded Data School (`http://dataschool.io`) in order to provide in-depth educational resources that are accessible to data science novices. He has an active YouTube channel (`http://youtube.com/dataschool`) and can also be found on Twitter (`@justmarkham`).

Alberto Gonzalez Paje is an economist specializing in information management systems and data science. Educated in Spain and the Netherlands, he has developed an international career as a data analyst at companies such as Coca Cola, Accenture, Bestiario, and CartoDB. He focuses on business strategy, planning, control, and data analysis. He loves architecture, cartography, the Mediterranean way of life, and sports.

Bastiaan Sjardin is a data scientist and entrepreneur with a background in artificial intelligence, mathematics, and machine learning. He has an MSc degree in cognitive science and mathematical statistics at the University of Leiden. In the past 5 years, he has worked on a wide range of data science projects. He is a frequent Community TA with Coursera for the "Social Network analysis" course at the University of Michigan. His programming language of choice is R and Python. Currently, he is the cofounder of Quandbee (`www.quandbee.com`), a company specialized in machine learning applications.

Michele Usuelli is a data scientist living in London, specializing in R and Hadoop. He has an MSc in mathematical engineering and statistics, and he has worked in fast-paced, growing environments, such as a big data start-up in Milan, the new pricing and analytics division of a big publishing company, and a leading R-based company. He is the author of *R Machine Learning Essentials*, *Packt Publishing*, which is a book that shows how to solve business challenges with data-driven solutions. He has also written articles on R-bloggers and is active on StackOverflow.

Zacharias Voulgaris, PhD, is a data scientist with machine learning expertise. His first degree was in production engineering and management, while his post-graduate studies focused on information systems (MSc) and machine learning (PhD). He has worked as a researcher at Georgia Tech and as a data scientist at Elavon Inc. He currently works for Microsoft as a program manager, and he is involved in a variety of big data projects in the field of web search. He has written several research papers and a number of web articles on data science-related topics and has authored his own book titled *Data Scientist: The Definite Guide to Becoming a Data Scientist*.

Table of Contents

Preface

"*A journey of a thousand miles begins with a single step.*"

–Laozi (604 BC - 531 BC)

Data science is a relatively new knowledge domain that requires the successful integration of linear algebra, statistical modelling, visualization, computational linguistics, graph analysis, machine learning, business intelligence, and data storage and retrieval.

The Python programming language, having conquered the scientific community during the last decade, is now an indispensable tool for the data science practitioner and a must-have tool for every aspiring data scientist. Python will offer you a fast, reliable, cross-platform, mature environment for data analysis, machine learning, and algorithmic problem solving. Whatever stopped you before from mastering Python for data science applications will be easily overcome by our easy step-by-step and example-oriented approach that will help you apply the most straightforward and effective Python tools to both demonstrative and real-world datasets.

Leveraging your existing knowledge of Python syntax and constructs (but don't worry, we have some Python tutorials if you need to acquire more knowledge on the language), this book will start by introducing you to the process of setting up your essential data science toolbox. Then, it will guide you through all the data munging and preprocessing phases. A necessary amount of time will be spent in explaining the core activities related to transforming, fixing, exploring, and processing data. Then, we will demonstrate advanced data science operations in order to enhance critical information, set up an experimental pipeline for variable and hypothesis selection, optimize hyper-parameters, and use cross-validation and testing in an effective way.

Finally, we will complete the overview by presenting you with the main machine learning algorithms, graph analysis technicalities, and all the visualization instruments that can make your life easier when it comes to presenting your results.

In this walkthrough, which is structured as a data science project, you will always be accompanied by clear code and simplified examples to help you understand the underlying mechanics and real-world datasets. It will also give you hints dictated by experience to help you immediately operate on your current projects. Are you ready to start? We are sure that you are ready to take the first step towards a long and incredibly rewarding journey.

What this book covers

Chapter 1, *First Steps*, introduces you to all the basic tools (command shell for interactive computing, libraries, and datasets) necessary to immediately start on data science using Python.

Chapter 2, *Data Munging*, explains how to upload the data to be analyzed by applying alternative techniques when the data is too big for the computer to handle. It introduces all the key data manipulation and transformation techniques.

Chapter 3, *The Data Science Pipeline*, offers advanced explorative and manipulative techniques, enabling sophisticated data operations to create and reduce predictive features, spot anomalous cases and apply validation techniques.

Chapter 4, *Machine Learning*, guides you through the most important learning algorithms that are available in the Scikit-learn library, which demonstrates the practical applications and points out the key values to be checked and the parameters to be tuned in order to get the best out of each machine learning technique.

Chapter 5, *Social Network Analysis*, elaborates the practical and effective skills that are required to handle data that represents social relations or interactions.

Chapter 6, *Visualization*, completes the data science overview with basic and intermediate graphical representations. They are indispensable if you want to visually represent complex data structures and machine learning processes and results.

Chapter 7, *Strengthen Your Python Foundations*, covers a few Python examples and tutorials focused on the key features of the language that it is indispensable to know in order to work on data science projects.

This chapter is not part of the book, but it has to be downloaded from Packt Publishing website at `https://www.packtpub.com/sites/default/files/downloads/0429OS_Chapter-07.pdf`.

What you need for this book

Python and all the data science tools mentioned in the book, from IPython to Scikit-learn, are free of charge and can be freely downloaded from the Internet. To run the code that accompanies the book, you need a computer that uses Windows, Linux, or Mac OS operating systems. The book will introduce you step-by-step to the process of installing the Python interpreter and all the tools and data that you need to run the examples.

Who this book is for

This book builds on the core skills that you already have, enabling you to become an efficient data science practitioner. Therefore, it assumes that you know the basics of programming and statistics.

The code examples provided in the book won't require you to have a mastery of Python, but we will assume that you know at least the basics of Python scripting, lists and dictionary data structures, and how class objects work. Before starting, you can quickly acquire such skills by spending a few hours on the online courses that we are going to suggest in the first chapter. You can also use the tutorial provided on the Packt Publishing website.

No advanced data science concepts are necessary though, as we will provide you with the information that is essential to understand all the core concepts that are used by the examples in the book.

Summarizing, this book is for the following:

- Novice and aspiring data scientists with limited Python experience and a working knowledge of data analysis, but no specific expertise of data science algorithms
- Data analysts who are proficient in statistic modeling using R or MATLAB tools and who would like to exploit Python to perform data science operations
- Developers and programmers who intend to expand their knowledge and learn about data manipulation and machine learning

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "When inspecting the linear model, first check the `coef_` attribute."

A block of code is set as follows:

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

Since we will be using IPython Notebooks along most of the examples, expect to have always an input (marked as `In:`) and often an output (marked `Out:`) from the cell containing the block of code. On your computer you have just to input the code after the `In:` and check if results correspond to the `Out:` content:

```
In: clf.fit(X, y)
Out: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
degree=3, gamma=0.0, kernel='rbf', max_iter=-1, probability=False,
random_state=None, shrinking=True, tol=0.001, verbose=False)
```

When a command should be given in the terminal command line, you'll find the command with the prefix `$>`, otherwise, if it's for the Python REPL, it will be preceded by `>>>`:

```
$>python
>>> import sys
>>> print sys.version_info
```

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

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

Piracy

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

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

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

Questions

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

1
First Steps

Whether you are an eager learner of data science or a well-grounded data science practitioner, you can take advantage of this essential introduction to Python for data science. You can use it to the fullest if you already have at least some previous experience in basic coding, writing general-purpose computer programs in Python, or some other data analysis-specific language, such as MATLAB or R.

The book will delve directly into Python for data science, providing you with a straight and fast route to solve various data science problems using Python and its powerful data analysis and machine learning packages. The code examples that are provided in this book don't require you to master Python. However, they will assume that you at least know the basics of Python scripting, data structures such as lists and dictionaries, and the working of class objects. If you don't feel confident about this subject or have minimal knowledge of the Python language, we suggest that before you read this book, you should take an online tutorial, such as the Code Academy course at `http://www.codecademy.com/en/tracks/python` or Google's Python class at `https://developers.google.com/edu/python/`. Both the courses are free, and in a matter of a few hours of study, they should provide you with all the building blocks that will ensure that you enjoy this book to the fullest. We have also prepared a tutorial of our own, which you can download from the Packt Publishing website, in order to provide an integration of the two aforementioned free courses.

In any case, don't be intimidated by our starting requirements; mastering Python for data science applications isn't as arduous as you may think. It's just that we have to assume some basic knowledge on the reader's part because our intention is to go straight to the point of using data science without having to explain too much about the general aspects of the language that we will be using.

Are you ready, then? Let's start!

In this short introductory chapter, we will work out the basics to set off in full swing and go through the following topics:

- How to set up a Python **Data Science Toolbox**
- Using IPython
- An overview of the data that we are going to study in this book

Introducing data science and Python

Data science is a relatively new knowledge domain, though its core components have been studied and researched for many years by the computer science community. These components include linear algebra, statistical modelling, visualization, computational linguistics, graph analysis, machine learning, business intelligence, and data storage and retrieval.

Being a new domain, you have to take into consideration that currently the frontier of data science is still somewhat blurred and dynamic. Because of its various constituent set of disciplines, please keep in mind that there are different profiles of data scientists, depending on their competencies and areas of expertise.

In such a situation, what can be the best tool of the trade that you can learn and effectively use in your career as a data scientist? We believe that the best tool is Python, and we intend to provide you with all the essential information that you will need for a fast start.

Also, other tools such as R and MATLAB provide data scientists with specialized tools to solve specific problems in statistical analysis and matrix manipulation in data science. However, only Python completes your data scientist skill set. This multipurpose language is suitable for both development and production alike and is easy to learn and grasp, no matter what your background or experience is.

Created in 1991 as a general-purpose, interpreted, object-oriented language, Python has slowly and steadily conquered the scientific community and grown into a mature ecosystem of specialized packages for data processing and analysis. It allows you to have uncountable and fast experimentations, easy theory developments, and prompt deployments of scientific applications.

At present, the Python characteristics that render it an indispensable data science tool are as follows:

- Python can easily integrate different tools and offer a truly unifying ground for different languages (Java, C, Fortran, and even language primitives), data strategies, and learning algorithms that can be easily fitted together and which can concretely help data scientists forge new powerful solutions.

- It offers a large, mature system of packages for data analysis and machine learning. It guarantees that you will get all that you may need in the course of a data analysis, and sometimes even more.
- It is very versatile. No matter what your programming background or style is (object-oriented or procedural), you will enjoy programming with Python.
- It is cross-platform; your solutions will work perfectly and smoothly on Windows, Linux, and Mac OS systems. You won't have to worry about portability.
- Although interpreted, it is undoubtedly fast compared to other mainstream data analysis languages such as R and MATLAB (though it is not comparable to C, Java, and the newly emerged Julia language). It can be even faster, thanks to some easy tricks that we are going to explain in this book.
- It can work with in-memory big data because of its minimal memory footprint and excellent memory management. The memory garbage collector will often save the day when you load, transform, dice, slice, save, or discard data using the various iterations and reiterations of data wrangling.
- It is very simple to learn and use. After you grasp the basics, there's no other better way to learn more than by immediately starting with the coding.

Installing Python

First of all, let's proceed to introduce all the settings you need in order to create a fully working data science environment to test the examples and experiment with the code that we are going to provide you with.

Python is an open source, object-oriented, cross-platform programming language that, compared to its direct competitors (for instance, C++ and Java), is very concise. It allows you to build a working software prototype in a very short time. Did it become the most used language in the data scientist's toolbox just because of this? Well, no. It's also a general-purpose language, and it is very flexible indeed due to a large variety of available packages that solve a wide spectrum of problems and necessities.

Python 2 or Python 3?

There are two main branches of Python: 2 and 3. Although the third version is the newest, the *older* one is still the most used version in the scientific area, since a few libraries (see `http://py3readiness.org` for a compatibility overview) won't run otherwise. In fact, if you try to run some code developed for Python 2 with a Python 3 interpreter, it won't work. Major changes have been made to the newest version, and this has impacted past compatibility. So, please remember that there is no backward compatibility between Python 3 and 2.

In this book, in order to address a larger audience of readers and practitioners, we're going to adopt the Python 2 syntax for all our examples (at the time of writing this book, the latest release is 2.7.8). Since the differences amount to really minor changes, advanced users of Python 3 are encouraged to adapt and optimize the code to suit their favored version.

Step-by-step installation

Novice data scientists who have never used Python (so, we figured out that they don't have it readily installed on their machines) need to first download the installer from the main website of the project, `https://www.python.org/downloads/`, and then install it on their local machine.

> This section provides you with full control over what can be installed on your machine. This is very useful when you have to set up single machines to deal with different tasks in data science. Anyway, please be warned that a step-by-step installation really takes time and effort. Instead, installing a ready-made scientific distribution will lessen the burden of installation procedures and it may be well suited for first starting and learning because it saves you time and sometimes even trouble, though it will put a large number of packages (and we won't use most of them) on your computer all at once. Therefore, if you want to start immediately with an easy installation procedure, just skip this part and proceed to the next section, *Scientific distributions*.

Being a multiplatform programming language, you'll find installers for machines that either run on Windows or Unix-like operating systems. Please remember that some Linux distributions (such as Ubuntu) have Python 2 packeted in the repository, which makes the installation process even easier.

1. To open a python shell, type `python` in the terminal or click on the Python icon.
2. Then, to test the installation, run the following code in the Python interactive shell or REPL:

   ```
   >>> import sys
   >>> print sys.version_info
   ```

3. If a syntax error is raised, it means that you are running Python 3 instead of Python 2. Otherwise, if you don't experience an error and you can read that your Python version has the `attribute major=2`, then congratulations for running the right version of Python. You're now ready to move forward.

To clarify, when a command is given in the terminal command line, we prefix the command with `$>`. Otherwise, if it's for the Python REPL, it's preceded by `>>>`.

A glance at the essential Python packages

We mentioned that the two most relevant Python characteristics are its ability to integrate with other languages and its mature package system that is well embodied by PyPI (the Python Package Index; `https://pypi.python.org/pypi`), a common repository for a majority of Python packages.

The packages that we are now going to introduce are strongly analytical and will offer a complete Data Science Toolbox made up of highly optimized functions for working, optimal memory configuration, ready to achieve scripting operations with optimal performance. A walkthrough on how to install them is given in the following section.

Partially inspired by similar tools present in R and MATLAB environments, we will together explore how a few selected Python commands can allow you to efficiently handle data and then explore, transform, experiment, and learn from the same without having to write too much code or reinvent the wheel.

NumPy

NumPy, which is Travis Oliphant's creation, is the true analytical workhorse of the Python language. It provides the user with multidimensional arrays, along with a large set of functions to operate a multiplicity of mathematical operations on these arrays. Arrays are blocks of data arranged along multiple dimensions, which implement mathematical vectors and matrices. Arrays are useful not just for storing data, but also for fast matrix operations (vectorization), which are indispensable when you wish to solve ad hoc data science problems.

- **Website**: `http://www.numpy.org/`
- **Version at the time of print**: 1.9.1
- **Suggested install command**: `pip install numpy`

As a convention largely adopted by the Python community, when importing NumPy, it is suggested that you alias it as np:

```
import numpy as np
```

We will be doing this throughout the course of this book.

SciPy

An original project by Travis Oliphant, Pearu Peterson, and Eric Jones, SciPy completes NumPy's functionalities, offering a larger variety of scientific algorithms for linear algebra, sparse matrices, signal and image processing, optimization, fast Fourier transformation, and much more.

- **Website**: `http://www.scipy.org/`
- **Version at time of print**: 0.14.0
- **Suggested install command**: `pip install scipy`

pandas

The pandas package deals with everything that NumPy and SciPy cannot do. Thanks to its specific object data structures, DataFrames and Series, pandas allows you to handle complex tables of data of different types (which is something that NumPy's arrays cannot do) and time series. Thanks to Wes McKinney's creation, you will be able to easily and smoothly load data from a variety of sources. You can then slice, dice, handle missing elements, add, rename, aggregate, reshape, and finally visualize this data at your will.

- **Website**: `http://pandas.pydata.org/`
- **Version at the time of print**: 0.15.2
- **Suggested install command**: `pip install pandas`

Conventionally, pandas is imported as `pd`:

```
import pandas as pd
```

Scikit-learn

Started as part of the SciKits (SciPy Toolkits), Scikit-learn is the core of data science operations on Python. It offers all that you may need in terms of data preprocessing, supervised and unsupervised learning, model selection, validation, and error metrics. Expect us to talk at length about this package throughout this book. Scikit-learn started in 2007 as a Google Summer of Code project by David Cournapeau. Since 2013, it has been taken over by the researchers at INRA (French Institute for Research in Computer Science and Automation).

- **Website**: `http://scikit-learn.org/stable/`
- **Version at the time of print**: 0.15.2
- **Suggested install command**: `pip install scikit-learn`

Note that the imported module is named sklearn.

IPython

A scientific approach requires the fast experimentation of different hypotheses in a reproducible fashion. IPython was created by Fernando Perez in order to address the need for an interactive Python command shell (which is based on shell, web browser, and the application interface), with graphical integration, customizable commands, rich history (in the JSON format), and computational parallelism for an enhanced performance. IPython is our favored choice throughout this book, and it is used to clearly and effectively illustrate operations with scripts and data and the consequent results.

- **Website**: `http://ipython.org/`
- **Version at the time of print**: 2.3
- **Suggested install** command: `pip install "ipython[notebook]"`

Matplotlib

Originally developed by John Hunter, matplotlib is the library that contains all the building blocks that are required to create quality plots from arrays and to visualize them interactively.

You can find all the MATLAB-like plotting frameworks inside the pylab module.

- **Website**: `http://matplotlib.org/`
- **Version at the time of print**: 1.4.2
- **Suggested install** command: `pip install matplotlib`

You can simply import what you need for your visualization purposes with the following command:

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

Downloading the example code

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

Statsmodels

Previously part of SciKits, statsmodels was thought to be a complement to SciPy statistical functions. It features generalized linear models, discrete choice models, time series analysis, and a series of descriptive statistics as well as parametric and nonparametric tests.

- **Website**: `http://statsmodels.sourceforge.net/`
- **Version at the time of print**: 0.6.0
- **Suggested install command**: `pip install statsmodels`

Beautiful Soup

Beautiful Soup, a creation of Leonard Richardson, is a great tool to scrap out data from HTML and XML files retrieved from the Internet. It works incredibly well, even in the case of *tag soups* (hence the name), which are collections of malformed, contradictory, and incorrect tags. After choosing your parser (basically, the HTML parser included in Python's standard library works fine), thanks to Beautiful Soup, you can navigate through the objects in the page and extract text, tables, and any other information that you may find useful.

- **Website**: `http://www.crummy.com/software/BeautifulSoup/`
- **Version at the time of print**: 4.3.2
- **Suggested install command**: `pip install beautifulsoup4`

> Note that the imported module is named `bs4`.

NetworkX

Developed by the Los Alamos National Laboratory, NetworkX is a package specialized in the creation, manipulation, analysis, and graphical representation of real-life network data (it can easily operate with graphs made up of a million nodes and edges). Besides specialized data structures for graphs and fine visualization methods (2D and 3D), it provides the user with many standard graph measures and algorithms, such as the shortest path, centrality, components, communities, clustering, and PageRank. We will frequently use this package in *Chapter 5, Social Network Analysis*.

- **Website**: `https://networkx.github.io/`
- **Version at the time of print**: 1.9.1
- **Suggested install command**: `pip install networkx`

Conventionally, NetworkX is imported as `nx`:

```
import networkx as nx
```

NLTK

The **Natural Language Toolkit** (**NLTK**) provides access to corpora and lexical resources and to a complete suit of functions for statistical **Natural Language Processing** (**NLP**), ranging from tokenizers to part-of-speech taggers and from tree models to named-entity recognition. Initially, the package was created by Steven Bird and Edward Loper as an NLP teaching infrastructure for CIS-530 at the University of Pennsylvania. It is a fantastic tool that you can use to prototype and build NLP systems.

- **Website**: `http://www.nltk.org/`
- **Version at the time of print**: 3.0
- **Suggested install command**: `pip install nltk`

Gensim

Gensim, programmed by Radim Řehůřek, is an open source package that is suitable for the analysis of large textual collections with the help of parallel distributable online algorithms. Among advanced functionalities, it implements **Latent Semantic Analysis** (**LSA**), topic modeling by **Latent Dirichlet Allocation** (**LDA**), and Google's *word2vec*, a powerful algorithm that transforms text into vector features that can be used in supervised and unsupervised machine learning.

- **Website**: `http://radimrehurek.com/gensim/`
- **Version at the time of print**: 0.10.3
- **Suggested install command**: `pip install gensim`

PyPy

PyPy is not a package; it is an alternative implementation of Python 2.7.8 that supports most of the commonly used Python standard packages (unfortunately, NumPy is currently not fully supported). As an advantage, it offers enhanced speed and memory handling. Thus, it is very useful for heavy duty operations on large chunks of data and it should be part of your big data handling strategies.

- **Website**: `http://pypy.org/`
- **Version at time of print**: 2.4.0
- **Download page**: `http://pypy.org/download.html`

The installation of packages

Python won't come bundled with all you need, unless you take a specific premade distribution. Therefore, to install the packages you need, you can either use `pip` or `easy_install`. These are the two tools that run in the command line and make the process of installation, upgrade, and removal of Python packages a breeze. To check which tools have been installed on your local machine, run the following command:

```
$> pip
```

Alternatively, you can also run the following command:

```
$> easy_install
```

If both these commands end with an error, you need to install any one of them. We recommend that you use pip because it is thought of as an improvement over `easy_install`. By the way, packages installed by `pip` can be uninstalled and if, by chance, your package installation fails, `pip` will leave your system clean.

To install pip, follow the instructions given at `https://pip.pypa.io/en/latest/installing.html`.

The most recent versions of Python should already have pip installed by default. So, you may have it already installed on your system. If not, the safest way is to download the `get-pi.py` script from `https://bootstrap.pypa.io/get-pip.py` and then run it using the following:

```
$> python get-pip.py
```

The script will also install the setup tool from `https://pypi.python.org/pypi/setuptools`, which also contains `easy_install`.

You're now ready to install the packages you need in order to run the examples provided in this book. To install the generic package `<pk>`, you just need to run the following command:

```
$> pip install <pk>
```

Alternatively, you can also run the following command:

```
$> easy_install <pk>
```

After this, the package `<pk>` and all its dependencies will be downloaded and installed. If you're not sure whether a library has been installed or not, just try to import a module inside it. If the Python interpreter raises an `ImportError` error, it can be concluded that the package has not been installed.

This is what happens when the NumPy library has been installed:

```
>>> import numpy
```

This is what happens if it's not installed:

```
>>> import numpy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named numpy
```

In the latter case, you'll need to first install it through `pip` or `easy_install`.

> Take care that you don't confuse packages with modules. With pip, you install a package; in Python, you import a module. Sometimes, the package and the module have the same name, but in many cases, they don't match. For example, the sklearn module is included in the package named Scikit-learn.

Finally, to search and browse the Python packages available for Python, take a look at `https://pypi.python.org`.

Package upgrades

More often than not, you will find yourself in a situation where you have to upgrade a package because the new version is either required by a dependency or has additional features that you would like to use. First, check the version of the library you have installed by glancing at the `__version__` attribute, as shown in the following example, `numpy`:

```
>>> import numpy
>>> numpy.__version__ # 2 underscores before and after
'1.9.0'
```

Now, if you want to update it to a newer release, say the 1.9.1 version, you can run the following command from the command line:

```
$> pip install -U numpy==1.9.1
```

Alternatively, you can also use the following command:

```
$> easy_install --upgrade numpy==1.9.1
```

Finally, if you're interested in upgrading it to the latest available version, simply run the following command:

```
$> pip install -U numpy
```

You can alternatively also run the following command:

```
$> easy_install --upgrade numpy
```

Scientific distributions

As you've read so far, creating a working environment is a time-consuming operation for a data scientist. You first need to install Python and then, one by one, you can install all the libraries that you will need (sometimes, the installation procedures may not go as smoothly as you'd hoped for earlier).

If you want to save time and effort and want to ensure that you have a fully working Python environment that is ready to use, you can just download, install, and use the scientific Python distribution. Apart from Python, they also include a variety of preinstalled packages, and sometimes, they even have additional tools and an IDE. A few of them are very well known among data scientists, and in the sections that follow, you will find some of the key features of each of these packages.

We suggest that you first promptly download and install a scientific distribution, such as Anaconda (which is the most complete one), and after practicing the examples in the book, decide to fully uninstall the distribution and set up Python alone, which can be accompanied by just the packages you need for your projects.

Anaconda

Anaconda (`https://store.continuum.io/cshop/anaconda`) is a Python distribution offered by Continuum Analytics that includes nearly 200 packages, which include NumPy, SciPy, pandas, IPython, Matplotlib, Scikit-learn, and NLTK. It's a cross-platform distribution that can be installed on machines with other existing Python distributions and versions, and its base version is free. Additional add-ons that contain advanced features are charged separately. Anaconda introduces `conda`, a binary package manager, as a command-line tool to manage your package installations. As stated on the website, Anaconda's goal is to provide enterprise-ready Python distribution for large-scale processing, predictive analytics and scientific computing.

Enthought Canopy

Enthought Canopy (`https://www.enthought.com/products/canopy/`) is a Python distribution by Enthought, Inc. It includes more than 70 preinstalled packages, which include NumPy, SciPy, Matplotlib, IPython, and pandas. This distribution is targeted at engineers, data scientists, quantitative and data analysts, and enterprises. Its base version is free (which is named Canopy Express), but if you need advanced features, you have to buy a front version. It's a multiplatform distribution and its command-line install tool is `canopy_cli`.

PythonXY

PythonXY (`https://code.google.com/p/pythonxy/`) is a free, open source Python distribution maintained by the community. It includes a number of packages, which include NumPy, SciPy, NetworkX, IPython, and Scikit-learn. It also includes Spyder, an interactive development environment inspired by the MATLAB IDE. The distribution is free. It works only on Microsoft Windows, and its command-line installation tool is pip.

WinPython

WinPython (`http://winpython.sourceforge.net`) is also a free, open-source Python distribution maintained by the community. It is designed for scientists, and includes many packages such as NumPy, SciPy, Matplotlib, and IPython. It also includes Spyder as an IDE. It is free and portable (you can put it in any directory, or even in a USB flash drive). It works only on Microsoft Windows, and its command-line tool is the **WinPython Package Manager** (**WPPM**).

Introducing IPython

IPython is a special tool for interactive tasks, which contains special commands that help the developer better understand the code that they are currently writing. These are the commands:

- `<object>? and <object>??`: This prints a detailed description (with `??` being even more verbose) of the `<object>`
- `%<function>`: This uses the special `<magic function>`

Let's demonstrate the usage of these commands with an example. We first start the interactive console with the `ipython` command that is used to run IPython, as shown here:

```
$> ipython
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
Type "copyright", "credits" or "license" for more information.
IPython 2.3.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra
details.
In [1]: obj1 = range(10)
```

Then, in the first line of code, which is marked by IPython as `[1]`, we create a list of 10 numbers (from 0 to 9), assigning the output to an object named `obj1`:

```
In [2]: obj1?
Type:        list
String form: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Length:      10
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
In [3]: %timeit x=100
10000000 loops, best of 3: 23.4 ns per loop
In [4]: %quickref
```

In the next line of code, which is numbered `[2]`, we inspect the `obj1` object using the IPython command `?`. IPython introspects the object, prints its details (`obj` is a list that contains the values `[1, 2, 3..., 9]` and elements), and finally prints some general documentation on lists. It's not the case in this example. However, for complex objects, the usage of `??`instead of `?`gives a more verbose output.

In line `[3]`, we use the magic function `timeit` to a Python assignment (`x=100`). The `timeit` function runs this instruction many times and stores the computational time needed to execute it. Finally, it prints the average time that was taken to run the Python function.

We complete the overview with a list of all the possible IPython special functions by running the helper function `quickref`, as shown in line `[4]`.

As you noticed, each time we use IPython, we have an input cell and optionally, an output cell, if there is something that has to be printed on `stdout`. Each input is numbered, so it can be referenced inside the IPython environment itself. For our purposes, we don't need to provide such references in the code of the book. Therefore, we will just report inputs and outputs without their numbers. However, we'll use the generic `In:` and `Out:` notations to point out the input and output cells. Just copy the commands after `In:` to your own IPython cell and expect an output that will be reported on the following `Out:`.

Therefore, the basic notations will be:

- The `In:` command
- The `Out:` output (wherever it is present and useful to be reported in the book)

Otherwise, if we expect you to operate directly on the Python console, we will use the following form:

```
 >>> command
```

Wherever necessary, the command-line input and output will be written as follows:

```
$> command
```

Moreover, to run the `bash` command in the IPython console, prefix it with a "`!`" (an exclamation mark):

```
In: !ls
Applications    Google Drive    Public          Desktop
Develop
Pictures        env             temp
...
In: !pwd
/Users/mycomputer
```

The IPython Notebook

The main goal of the IPython Notebook is easy storytelling. Storytelling is essential in data science because you must have the power to do the following:

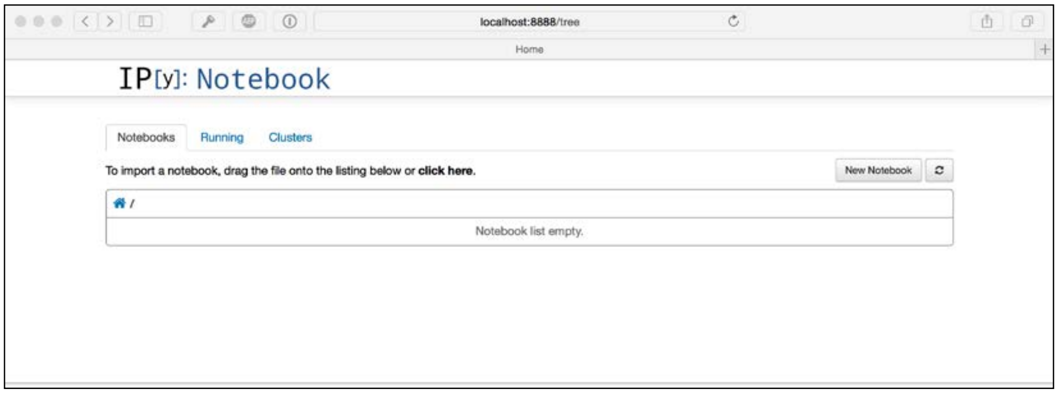
- See intermediate (debugging) results for each step of the algorithm you're developing
- Run only some sections (or cells) of the code
- Store intermediate results and have the ability to version them
- Present your work (this will be a combination of text, code, and images)

Here comes IPython; it actually implements all the preceding actions.

1. To launch the IPython Notebook, run the following command:

   ```
   $> ipython notebook
   ```

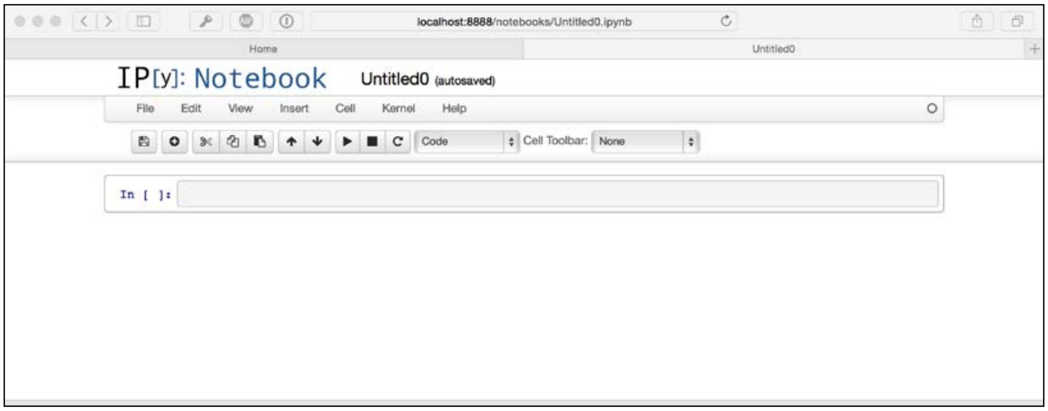
2. A web browser window will pop up on your desktop, backed by an IPython server instance. This is the how the main window looks:



To import a notebook, drag the file onto the listing below or **click here.**

Notebook list empty.

3. Then, click on **New Notebook**. A new window will open, as shown in the following screenshot:



This is the web app that you'll use to compose your story. It's very similar to a Python IDE, with the bottom section (where you can write the code) composed of cells.

A cell can be either a piece of text (eventually formatted with a markup language) or a piece of code. In the second case, you have the ability to run the code, and any eventual output (the standard output) will be placed under the cell. The following is a very simple example of the same:

```
In: import random
         a = random.randint(0, 100)
         a
Out: 16
In: a*2
Out: 32
```

In the first cell, which is denoted by `In:`, we import the random module, assign a random value between 0 and 100 to the variable `a`, and print the value. When this cell is run, the output, which is denoted as `Out:`, is the random number. Then, in the next cell, we will just print the double of the value of the variable `a`.

As you can see, it's a great tool to debug and decide which parameter is best for a given operation. Now, what happens if we run the code in the first cell? Will the output of the second cell be modified since `a` is different? Actually, no. Each cell is independent and autonomous. In fact, after we run the code in the first cell, we fall in this inconsistent status:

```
In: import random
         a = random.randint(0, 100)
         a
Out: 56
In: a*2
Out: 32
```

> Also note that the number in the squared parenthesis has changed (from 1 to 3) since it's the third executed command (and its output) from the time the notebook started. Since each cell is autonomous, by looking at these numbers, you can understand their order of execution.

IPython is a simple, flexible, and powerful tool. However, as seen in the preceding example, you must note that when you update a variable that is going to be used later on in your Notebook, remember to run all the cells following the updated code so that you have a consistent state.

When you save an IPython notebook, the resulting `.ipynb` file is JSON formatted, and it contains all the cells and their content, plus the output. This makes things easier because you don't need to run the code to see the notebook (actually, you also don't need to have Python and its set of toolkits installed). This is very handy, especially when you have pictures featured in the output and some very time-consuming routines in the code. A downside of using the IPython Notebook is that its file format, which is JSON structured, cannot be easily read by humans. In fact, it contains images, code, text, and so on.

Now, let's discuss a data science related example (don't worry about understanding it completely):

```
In:
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
```

In the following cell, some Python modules are imported:

```
In:
boston_dataset = datasets.load_boston()
X_full = boston_dataset.data
Y = boston_dataset.target
print X_full.shape
print Y.shape
Out:
(506, 13)
(506,)
```

Then, in `cell [2]`, the dataset is loaded and an indication of its shape is shown. The dataset contains 506 house values that were sold in the suburbs of Boston, along with their respective data arranged in columns. Each column of the data represents a feature. A feature is a characteristic property of the observation. Machine learning uses features to establish models that can turn them into predictions. If you are from a statistical background, you can add features that can be intended as variables (values that vary with respect to the observations).

To see a complete description of the dataset, `print boston_dataset.DESCR`.

After loading the observations and their features, in order to provide a demonstration of how IPython can effectively support the development of data science solutions, we will perform some transformations and analysis on the dataset. We will use classes, such as `SelectKBest`, and methods, such as `.getsupport()` or `.fit()`. Don't worry if these are not clear to you now; they will all be covered extensively later in this book. Try to run the following code:

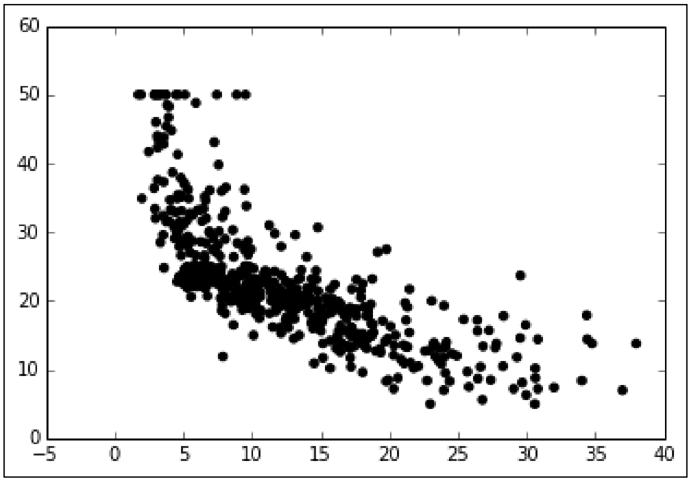
```
In:
selector = SelectKBest(f_regression, k=1)
```

```
selector.fit(X_full, Y)
X = X_full[:, selector.get_support()]
print X.shape
Out:
(506, 1)
```

In:, we select a feature (the most discriminative one) of the SelectKBest class that is fitted to the data by using the .fit() method. Thus, we reduce the dataset to a vector with the help of a selection operated by indexing on all the rows and on the selected feature, which can be retrieved by the .get_support() method.

Since the target value is a vector, we can, therefore, try to see whether there is a linear relation between the input (the feature) and the output (the house value). When there is a linear relationship between two variables, the output will constantly react to changes in the input by the same proportional amount and direction.

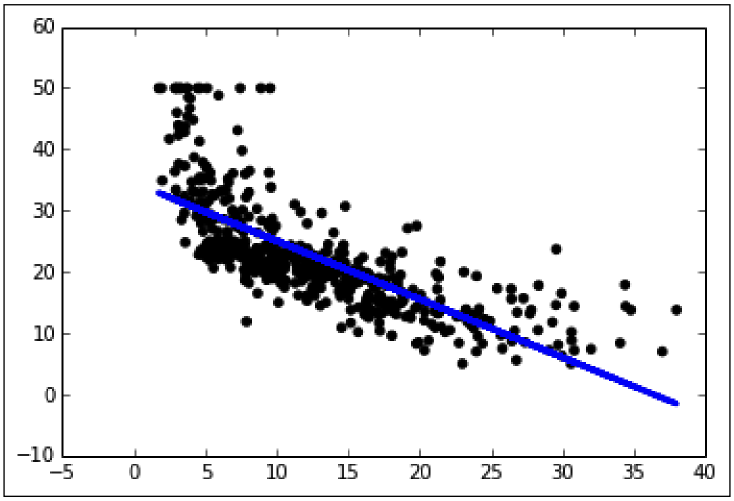
```
In:
plt.scatter(X, Y, color='black')
plt.show()
```



In our example, as *X* increases, *Y* decreases. However, this does not happen at a constant rate, because the rate of change is intense up to a certain *X* value but then it decreases and becomes constant. This is a condition of nonlinearity, and we can furthermore visualize it using a regression model. This model hypothesizes that the relationship between *X* and *Y* is linear in the form of $y=a+bX$. Its *a* and *b* parameters are estimated according to a certain criteria.

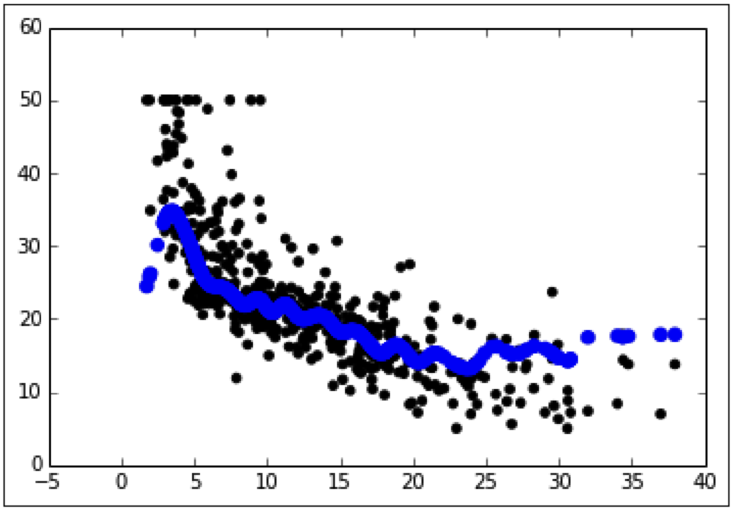
In the fourth cell, we scatter the input and output values for this problem:

```
In:
regressor = LinearRegression(normalize=True)
regressor.fit(X, Y)
plt.scatter(X, Y, color='black')
plt.plot(X, regressor.predict(X), color='blue', linewidth=3)
plt.show()
```

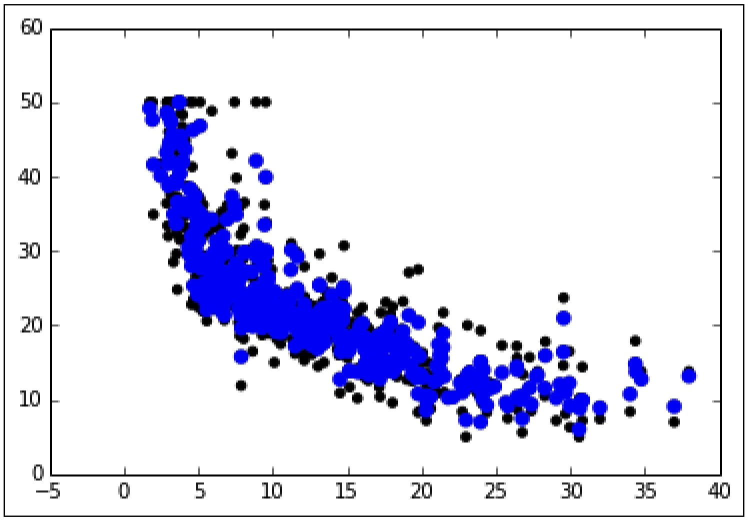


In the next cell, we create a regressor (a simple linear regression with feature normalization), train the regressor, and finally plot the best linear relation (that's the linear model of the regressor) between the input and output. Clearly, the linear model is an approximation that is not working well. We have two possible roads that we can follow at this point. We can transform the variables in order to make their relationship linear, or we can use a nonlinear model. **Support Vector Machine** (**SVM**) is a class of models that can easily solve nonlinearities. Also, **Random Forests** is another model for the automatic solving of similar problems. Let's see them in action in IPython:

```
In:
regressor = SVR()
regressor.fit(X, Y)
plt.scatter(X, Y, color='black')
plt.scatter(X, regressor.predict(X), color='blue', linewidth=3)
plt.show()
```



```
In:
regressor = RandomForestRegressor()
regressor.fit(X, Y)
plt.scatter(X, Y, color='black');
plt.scatter(X, regressor.predict(X), color='blue', linewidth=3)
plt.show()
```



Finally, in the last two cells, we will repeat the same procedure. This time we will use two nonlinear approaches: an SVM and a Random Forest based regressor.

Having been written down on the IPython interface, this demonstrative code solves the nonlinearity problem. At this point, it is very easy to change the selected feature, regressor, the number of features we use to train the model, and so on, by simply modifying the cells where the script is. Everything can be done interactively, and according to the results we see, we can decide both what should be kept or changed and what is to be done next.

Datasets and code used in the book

As we progress through the concepts presented in this book, in order to facilitate the reader's understanding, learning, and memorizing processes, we will illustrate practical and effective data science Python applications on various explicative datasets. The reader will always be able to immediately replicate, modify, and experiment with the proposed instructions and scripts on the data that we will use in this book.

As for the code that you are going to find in this book, we will limit our discussions to the most essential commands in order to inspire you from the beginning of your data science journey with Python to do more with less by leveraging key functions from the packages we presented beforehand.

Given our previous introduction, we will present the code to be run interactively as it appears on an IPython console or Notebook.

All the presented code will be offered in Notebooks, which is available on the Packt Publishing website (as pointed out in the Preface). As for the data, we will provide different examples of datasets.

Scikit-learn toy datasets

The Scikit-learn toy dataset is embedded in the Scikit-learn package. Such datasets can easily be directly loaded into Python by the import command, and they don't require any download from any external Internet repository. Some examples of this type of dataset are the Iris, Boston, and Digits datasets, to name the principal ones mentioned in uncountable publications and books, and a few other classic ones for classification and regression.

Structured in a dictionary-like object, besides the features and target variables, they offer complete descriptions and contextualization of the data itself.

For instance, to load the Iris dataset, enter the following commands:

```
In: from sklearn import datasets
In: iris = datasets.load_iris()
```

After loading, we can explore the data description and understand how the features and targets are stored. Basically, all Scikit-learn datasets present the following methods:

- `.DESCR`: This provides a general description of the dataset
- `.data`: This contains all the features
- `.feature_names`: This reports the names of the features
- `.target`: This contains the target values expressed as values or numbered classes
- `.target_names`: This reports the names of the classes in the target
- `.shape`: This is a method that you can apply to both `.data` and `.target`; it reports the number of observations (the first value) and features (the second value, if present) that are present

Now, let's just try to implement them (no output is reported, but the print commands will provide you with plenty of information):

```
In: print iris.DESCR
In: print iris.data
In: print iris.data.shape
In: print iris.feature_names
In: print iris.target
In: print iris.target.shape
In: print iris.target_names
```

Now, you should know something more about the dataset—about how many examples and variables are present and what their names are.

Notice that the main data structures that are enclosed in the iris object are the two arrays, data and target:

```
In: print type(iris.data)
Out: <type 'numpy.ndarray'>
```

`Iris.data` offers the numeric values of the variables named `sepal length`, `sepal width`, `petal length`, and `petal width` arranged in a matrix form (150,4), where 150 is the number of observations and 4 is the number of features. The order of the variables is the order presented in `iris.feature_names`.

`Iris.target` is a vector of integer values, where each number represents a distinct class (refer to the content of target_names; each class name is related to its index number and *setosa*, which is the zero element of the list, is represented as `0` in the target vector).

The `Iris flower` dataset was first used in 1936 by Ronald Fisher, who was one of the fathers of modern statistical analysis, in order to demonstrate the functionality of linear discriminant analysis on a small set of empirically verifiable examples (each of the 150 data points represented iris flowers). These examples were arranged into tree balanced species classes (each class consisted of one-third of the examples) and were provided with four metric descriptive variables that, when combined, were able to separate the classes.

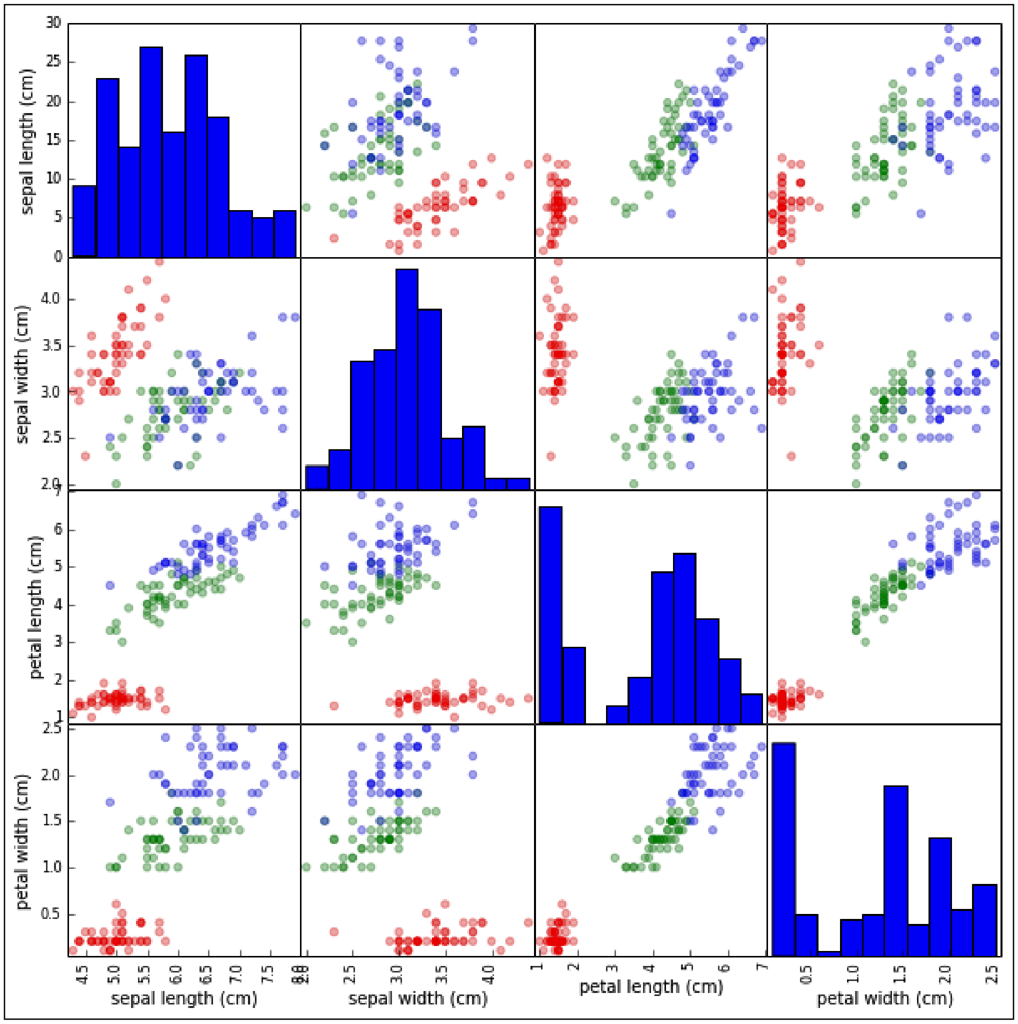
The advantage of using such a dataset is that it is very easy to load, handle, and explore for different purposes, from supervised learning to graphical representation. Modeling activities take almost no time on any computer, no matter what its specifications are. Moreover, the relationship between the classes and the role of the explicative variables are well known. So, the task is challenging, but it is not arduous.

For example, let's just observe how classes can be easily separated when you wish to combine at least two of the four available variables by using a scatterplot matrix.

Scatterplot matrices are arranged in a matrix format, whose columns and rows are the dataset variables. The elements of the matrix contain single scatterplots whose x values are determined by the row variable of the matrix and y values by the column variable. The diagonal elements of the matrix may contain a distribution histogram or some other univariate representation of the variable at the same time in its row and column.

The pandas library offers an off-the-shelf function to quickly make up scatterplot matrices and start exploring relationship and distributions between the quantitative variables in a dataset.

```
In:
import pandas as pd
import numpy as np
In: colors = list()
In: palette = {0: "red", 1: "green", 2: "blue"}
In:
for c in np.nditer(iris.target): colors.append(palette[int(c)])
    # using the palette dictionary, we convert
    # each numeric class into a color string
In: dataframe = pd.DataFrame(iris.data,
columns=iris.feature_names)
In: scatterplot = pd.scatter_matrix(dataframe, alpha=0.3,
figsize=(10, 10), diagonal='hist', color=colors, marker='o',
grid=True)
```



We encourage you to experiment a lot with this dataset and with similar ones before you work on other complex real data, because the advantage of focusing on an accessible, non-trivial data problem is that it can help you to quickly build your foundations on data science.

After a while, anyway, though useful and interesting for your learning activities, toy datasets will start limiting the variety of different experimentations that you can achieve. In spite of the insight provided, in order to progress, you'll need to gain access to complex and realistic data science topics. We will, therefore, have to resort to some external data.

The MLdata.org public repository

The second type of example dataset that we will present can be downloaded directly from the machine learning dataset repository, or from the **LIBSVM** data website. Contrary to the previous dataset, in this case, you will need to have access to the Internet.

First of all, `mldata.org` is a public repository for machine learning datasets that is hosted by the TU Berlin University and supported by **Pattern Analysis, Statistical Modelling, and Computational Learning** (**PASCAL**), a network funded by the European Union.

For example, if you need to download all the data related to earthquakes since 1972 as reported by the United States Geological Survey, in order to analyze the data to search for predictive patterns you will find the data repository at `http://mldata.org/repository/data/viewslug/global-earthquakes/` (here, you will find a detailed description of the data).

Note that the directory that contains the dataset is `global-earthquakes`; you can directly obtain the data using the following commands:

```
In: from sklearn.datasets import fetch_mldata
In: earthquakes = fetch_mldata('global-earthquakes')
In: print earthquakes.data
In: print earthquakes.data.shape
Out: (59209L, 4L)
```

As in the case of the Scikit-learn package toy dataset, the obtained object is a complex dictionary-like structure, where your predictive variables are `earthquakes.data` and your target to be predicted is `earthquakes.target`. This being the real data, in this case, you will have quite a lot of examples and just a few variables available.

LIBSVM data examples

LIBSVM Data (`http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`) is a page-gathering data from many other collections. It offers different regression, binary, and multilabel classification datasets stored in the LIBSVM format. This repository is quite interesting if you wish to experiment with the support vector machine's algorithm.

If you want to load a dataset, first go to the page where you wish to visualize the data. In this case, visit `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/a1a` and take down the address. Then, you can proceed by performing a direct download:

```
In: import urllib2
In: target_page =
'http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/a1a'
In: a2a = urllib2.urlopen(target_page)
In: from sklearn.datasets import load_svmlight_file
In: X_train, y_train = load_svmlight_file(a2a)
In: print X_train.shape, y_train.shape
Out: (2265, 119) (2265L,)
```

In return, you will get two single objects: a set of training examples in a sparse matrix format and an array of responses.

Loading data directly from CSV or text files

Sometimes, you may have to download the datasets directly from their repository using a web browser or a `wget` command.

If you have already downloaded and unpacked the data (if necessary) into your working directory, the simplest way to load your data and start working is offered by the NumPy and the pandas library with their respective `loadtxt` and `read_csv` functions.

For instance, if you intend to analyze the Boston housing data and use the version present at `http://mldata.org/repository/data/viewslug/regression-datasets-housing/`, you first have to download the `regression-datasets-housing.csv` file in your local directory.

Since the variables in the dataset are all numeric (13 continuous and one binary), the fastest way to load and start using it is by trying out the NumPy function `loadtxt` and directly loading all the data into an array.

Even in real-life datasets, you will often find mixed types of variables, which can be addressed by `pandas.read_table` or `pandas.read_csv`. Data can then be extracted by the `values` method; `loadtxt` can save a lot of memory if your data is already numeric since it does not require any in-memory duplication.

```
In: housing = np.loadtxt('regression-datasets-
housing.csv',delimiter=',')
In: print type(housing)
Out: <type 'numpy.ndarray'>
In: print housing.shape
Out:(506L, 14L)
```

The loadtxt function expects, by default, tabulation as a separator between the values on a file. If the separator is a colon (,) or a semi-colon(;), you have to explicit it using the parameter delimiter.

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

```
>>> type(np.loadtxt)
<type 'function'>
```

```
>>> help(np.loadtxt)
```

Help on function loadtxt in module numpy.lib.npyio.

Another important default parameter is dtype, which is set to float.

> This means that loadtxt will force all the loaded data to be converted into a floating point number.

If you need to determinate a different type (for example, an int), you have to declare it beforehand.

For instance, if you want to convert numeric data to int, use the following code:

```
In: housing_int = np.loadtxt('regression-datasets-
housing.csv',delimiter=',', dtype=int)
```

Printing the first three elements of the row of the housing and housing_int arrays can help you understand the difference:

```
In: print housing[0,:3], '\n', housing_int[0,:3]
```

```
Out:
```

```
[  6.32000000e-03   1.80000000e+01   2.31000000e+00]
```

```
[ 0 18  2]
```

Frequently, though not always the case in our example, the data on files feature in the first line a textual header that contains the name of the variables. In this situation, the parameter that is skipped will point out the row in the loadtxt file from where it will start reading the data. Being the header on row 0 (in Python, counting always starts from 0), parameter skip=1 will save the day and allow you to avoid an error and fail to load your data.

The situation would be slightly different if you were to download the Iris dataset, which is present at http://mldata.org/repository/data/viewslug/datasets-uci-iris/. In fact, this dataset presents a qualitative target variable, class, which is a string that expresses the iris species. Specifically, it's a categorical variable with four levels.

Therefore, if you were to use the loadtxt function, you will get a value error due to the fact that an array must have all its elements of the same type. The variable class is string, whereas the other variables are constituted of floating point values.

How to proceed? The pandas library offers the solution, thanks to its DataFrame data structure that can easily handle datasets in a matrix form (row per columns) that is made up of different types of variables.

First of all, just download the datasets-uci-iris.csv file and have it saved in your local directory.

At this point, using pandas' read_csv is quite straightforward:

```
In: iris_filename = 'datasets-uci-iris.csv'

In: iris = pd.read_csv(iris_filename, sep=',', decimal='.',
header=None, names= ['sepal_length', 'sepal_width', 'petal_length',
'petal_width', 'target'])

In: print type(iris)

Out: <class 'pandas.core.frame.DataFrame'>
```

Apart from the filename, you can specify the separator (sep), the way the decimal points are expressed (decimal), whether there is a header (in this case, header=None; normally, if you have a header, then header=0), and the name of the variable—where there is one (you can use a list; otherwise, pandas will provide some automatic naming).

> Also, we have defined names that use single words (instead of spaces, we used underscores). Thus, we can later directly extract single variables by calling them as we do for methods; for instance, iris.sepal_length will extract the sepal length data.

If, at this point, you need to convert the pandas DataFrame into a couple of NumPy arrays that contain the data and target values, this can be easily done in a couple of commands:

```
In: iris_data = iris.values[:,:4]

In: iris_target, iris_target_labels = pd.factorize(iris.target)

In: print iris_data.shape, iris_target.shape

Out: (150L, 4L) (150L,)
```

Scikit-learn sample generators

As a last learning resource, Scikit-learn also offers the possibility to quickly create synthetic datasets for regression, binary and multilabel classification, cluster analysis, and dimensionality reduction.

The main advantage of recurring to synthetic data lies in its instantaneous creation in the working memory of your Python console. It is, therefore, possible to create bigger data examples without having to engage in long downloading sessions from the Internet (and saving a lot of stuff on your disk).

For example, you may need to work on a million example classification problem:

```
In: from sklearn import datasets # We just import the "datasets" module
In: X,y = datasets.make_classification(n_samples=10**6,
n_features=10, random_state=101)
In: print X.shape,  y.shape
Out: (1000000L, 10L) (1000000L,)
```

After importing just the datasets module, we ask, using the make_classification command, for 1 million examples (the `n_samples` parameter) and 10 useful features (`n_features`). The `random_state` should be `101`, so we can be assured that we can replicate the same datasets at a different time and in a different machine.

For instance, you can type the following command:

```
$> datasets.make_classification(1, n_features=4, random_state=101)
```

This will always give you the following output:

```
(array([[-3.31994186, -2.39469384, -2.35882002,  1.40145585]]),
array([0]))
```

No matter what the computer and the specific situation is, `random_state` assures deterministic results that make your experimentations perfectly replicable.

Defining the `random_state` parameter using a specific integer number (in this case `101`, but it may be any number that you prefer or find useful) allows the easy replication of the same dataset on your machine, the way it is set up, on different operating systems, and on different machines.

By the way, did it take too long?

On a i3-2330M CPU @ 2.20GHz machine, it takes:

```
In: %timeit X,y = datasets.make_classification(n_samples=10**6,
n_features=10, random_state=101)

Out: 1 loops, best of 3: 2.17 s per loop
```

If it doesn't seem so also on your machine and if you are ready, having set up and tested everything up to this point, we can start our data science journey.

Summary

In this short introductory chapter, we installed everything that we will be using throughout this book, even the examples, which were installed either directly or by using a scientific distribution. We also introduced you to IPython and demonstrated how you can have access to the data run in the tutorials.

In the next chapter, *Data Munging*, we will have an overview of the data science pipeline and explore all the key tools to handle and prepare data before you apply any learning algorithm and set up your hypothesis experimentation schedule.

2
Data Munging

We are just getting into action! In this new chapter, you'll essentially learn how to munge data. What does munging data imply?

The term *munge* is a technical term coined about half a century ago by the students of the **Massachusetts Institute of Technology** (**MIT**). Munging means to change, in a series of well-specified and reversible steps, a piece of original data to a completely different (and hopefully a more useful) one. Deep-rooted in hacker culture, munging is often heard in data science processes in parallel with other almost completely synonymous terms, such as data wrangling and data preparation.

Given such premises, in this chapter, the following topics will be covered:

- The data science process (so you'll know what is going on and what's next)
- Uploading data from a file
- Selecting the data you need
- Cleaning up missing or wrong data
- Adding, inserting, and deleting data
- Grouping and transforming data to obtain new, meaningful information
- Managing to obtain a dataset matrix or an array to feed into the data science pipeline

The data science process

Although every data science project is different, for our illustrative purposes, we can partition them into a series of reduced and simplified phases.

The process starts with the obtaining of data, and this implies a series of possibilities, from simply uploading the data to assembling it from RDBMS or NoSQL repositories, and to synthetically generating it or scraping it from the web APIs or HTML pages.

Though this is a critical part of the data scientist's work, especially when faced with novel challenges, we will just briefly touch upon this aspect by offering the basic tools to get your data (even if it is too big) into your computer memory by using either a textual file present on your hard disk or the Web or using tables in RDBMS.

Then comes the data munging phase. Data will be inevitably always received in a form unsuitable for your analysis and experimentation. Thanks to a bunch of basic Python data structures and commands, you'll have to address all the problematic data in order to feed into your next phases a typical data matrix that has observations in rows and variables in columns.

Though less rewarding, data munging creates the foundations for every complex and sophisticated value-added analysis that you may have in mind to obtain.

Having completely defined the data matrix that you'll be working on, a new phase opens up, where you'll start observing your data and develop and test your hypothesis in a recurring loop. For instance, you'll explore your variables graphically. With the help of descriptive stats, you'll figure out how to create new variables by putting into action your domain knowledge. You'll address redundant and unexpected information (outliers, first of all) and select the most meaningful variables and effective parameters to be tested with a selection of machine learning algorithms.

You can figure out this phase, structured as a pipeline, where your data is repetitively processed until you have reached some meaningful result, which is most often represented by an error or optimization measure (that you will have to choose carefully). It may sometimes be represented by an interpretable insight that has to be verbally or visually described to your data science project's sponsors or other data scientists.

We won't ever get tired of remarking how everything starts with munging your data. Since even the longest journey starts with a single step, let's immediately step into this chapter.

Data loading and preprocessing with pandas

In the previous chapter, we discussed where to find useful datasets and examined basic import commands of Python packages. In this section, having kept your toolbox ready, you are about to learn how to structurally load, manipulate, preprocess, and polish data with pandas and NumPy.

Fast and easy data loading

Let's start with a CSV file and pandas. The pandas library offers the most accessible and complete function to load tabular data from a file (or a URL). By default, it will store the data into a specialized pandas data structure, index each row, separate variables by custom delimiters, infer the right data type for each column, convert data (if necessary), as well as parse dates, missing values, and erroneous values.

```
In: import pandas as pd

iris_filename = 'datasets-uci-iris.csv'

iris = pd.read_csv(iris_filename, sep=',', decimal='.', header=None,
names= ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'target'])
```

You can specify the name of the file, the character used as a separator (`sep`), the character used for the decimal placeholder (`decimal`), the character used if there is a header (`header`), and the variables names (using `names` and a list). The settings of the `sep=','` and `decimal='.'` parameters have default values, and they are redundant in function. Anyway, for the European style `csv`, it is important to point out both since in many countries, the separator and the decimal are different from the default ones.

If the dataset is not available online, you can follow these steps to download it from the Internet:

```
import urllib2

url = "http://aima.cs.berkeley.edu/data/iris.csv"

set1 = urllib2.Request(url)

iris_p = urllib2.urlopen(set1)

iris_other = pd.read_csv(iris_p, sep=',', decimal='.',
header=None, names= ['sepal_length', 'sepal_width',
'petal_length', 'petal_width', 'target'])
iris_other.head()
```

The resulting object, named iris, is a pandas DataFrame. It's more than a simple Python list or dictionary, and in the sections that follow, we will see some of its features. To get a rough idea of its content, you can print the first (or the last) row(s), using the following commands:

```
In: iris.head()
Out:
   sepal_length  sepal_width  petal_length  petal_width       target
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
In: iris.tail()
[...]
```

The function, if called without arguments, will print five lines. If you want a different number, just call the function with, as an argument, the number of rows you want to see, as follows:

```
In: iris.head(2)
```

The preceding command will print only the first two lines. Now, to get the names of the columns, you can simply use the following code:

```
In: iris.columns
Out: Index([u'sepal_length', u'sepal_width', u'petal_length',
  u'petal_width', u'target'], dtype='object')
```

The resulting object is a very interesting one. Apparently, it looks like a list, but it actually is a pandas Index. As you can suspect from the object's name, it indexes the columns' names. To extract the target column, for example, you can simply do the following:

```
In: Y = iris['target']
Y
Out:
0     Iris-setosa
1     Iris-setosa
2     Iris-setosa
```

```
3     Iris-setosa
...
149    Iris-virginica
Name: target, Length: 150, dtype: object
```

The type of the object `Y` is a pandas Series. Right now, think of it as a one-dimensional array with axis labels, as we will investigate it in depth later on. Now, we just understood that a pandas `Index` class acts like a dictionary index of the table's columns. Note that you can also get a list of columns through their indexes, as follows:

```
In: X = iris[['sepal_length', 'sepal_width']]
X
Out:
     sepal_length  sepal_width
0             5.1          3.5
1             4.9          3.0
2             4.7          3.2
...
147           6.5          3.0
148           6.2          3.4
149           5.9          3.0
[150 rows x 2 columns]
```

In this case, the result is a pandas DataFrame. Why is this difference observed when using the same function? Well, in the first case, we asked for a column. Therefore, the output was a 1-D vector (i.e. a pandas Series). In the second example, we asked for many columns and we obtained a matrix-like result (and we know that matrices are mapped as pandas DataFrames). A novice reader can simply spot the difference by looking at the heading of the output; if the columns are labelled, then you are dealing with a pandas DataFrame. On the other hand, if the result is a vector and has no heading, then that's a pandas Series.

So far, we have learned some fairly common steps from the data science process; after you load the dataset, you usually separate the features and target labels. Target labels are the ordinal numbers or textual strings that indicate the class associated with every set of features.

Then, the following step requires you to get an idea of the size of the problem, and therefore, you need to know the size of the dataset. Typically for each observation, we count a line, and for each feature, a column.

To get the dimension of the dataset, just use the attribute shape on both pandas DataFrame or Series, as shown in the following example:

```
In: X.shape
Out: (150, 5)
In:  Y.shape
Out: (150,)
```

The resulting object is a tuple that contains the size of the matrix/array in each dimension. Also note that pandas Series follows the same format (i.e. a tuple with only one element).

Dealing with problematic data

Now, you should be more confident with the basics of the process and be ready to face more problematic data since it is very common to have messy data in reality. Consequently, let's see what happens if the CSV file contains a header, and some missing values and dates. For example, to make things very easy and clear, let's imagine the situation of a travel agency. According to the temperature of three popular destinations, they record whether the user picks the first, second, or the third destination.

```
Date,Temperature_city_1,Temperature_city_2,Temperature_city_3,
  Which_destination
20140910,80,32,40,1
20140911,100,50,36,2
20140912,102,55,46,1
20140912,60,20,35,3
20140914,60,,32,3
20140914,,57,42,2
```

In this case, all the numbers are integers and the header is in the file. In our first attempt to load this dataset, we can give the following command:

```
In: import pandas as pd
In: fake_dataset = pd.read_csv('a_loading_example_1.csv', sep=',')
fake_dataset
Out:
  Date  Temperature_city_1  Temperature_city_2  Temperature_city_3
    Which_destination
0  20140910  80   32   40  1
1  20140911  100  50   36  2
```

```
2  20140912  102  55   46  1

3  20140913  60   20   35  3

4  20140914  60   NaN  32  3

5  20140915  NaN  57   42  2
```

That's a great achievement! pandas automatically named the columns with their actual name by taking the same from the first data row. We first detect a problem: all the data, even the dates, has been parsed as integers (or, in other cases, as string). If the format of the dates is not very strange, you can try the autodetection routines that specify the column that contains the date data. In this example, it works well with the following arguments:

```
In: fake_dataset = pd.read_csv('a_loading_example_1.csv',
parse_dates=[0])
fake_dataset

Out:

  Date  Temperature_city_1  Temperature_city_2  Temperature_city_3
Which_destination
0  2014-09-10  80  32  40  1

1  2014-09-11  100  50  36  2

2  2014-09-12  102  55  46  1

3  2014-09-13  60  20  35  3

4  2014-09-14  60  NaN  32  3

5  2014-09-15  NaN  57  42  2
```

Now, to get rid of the missing data that is indicated as `NaN`, replace them with a more meaningful number (let's say 50 Fahrenheit, for example). We can do this in the following way:

```
In: fake_dataset.fillna(50)

Out:

Date  Temperature_city_1  Temperature_city_2  Temperature_city_3  Which_
destination
0  2014-09-10  80  32  40  1

1  2014-09-11  100  50  36  2

2  2014-09-12  102  55  46  1

3  2014-09-13  60  20  35  3

4  2014-09-14  60  30  32  3

5  2014-09-15  30  57  42  2
```

Here, all the missing data disappears. Treating missing data can require different approaches. As an alternative to the previous command, values can be replaced by a negative constant value to mark the fact that they are different from others (and leave the guess for the learning algorithm):

```
In: fake_dataset.fillna(-1)
```

> Note that this method only fills missing values in view of the data (i.e. it doesn't modify the original DataFrame). In order to actually change them, use the `inplace=True argument`.

`NaN` values can also be replaced by the column mean or median value as a way to minimize the guessing error:

```
In: fake_dataset.fillna(fake_dataset.mean(axis=0))
```

The `.mean` method calculates the mean of the specified axis.

Please note that `axis= 0` implies the calculation of the means that span the rows so obtained means that they extend column-wise. Instead, `axis=1` spans columns and therefore, row-wise results are obtained. This works in the same way for all other methods that require the axis parameter, both in pandas and NumPy.

The `.median` method is analogous to `.mean`, but it computes the median value, which is useful in case the mean is not so well representative given too skewed data.

Another possible problem when handling real world datasets is the loading of a dataset with errors or bad lines. In this case, the default behavior of the `load_csv` method is to stop and raise an exception. A possible workaround, which is not always feasible, is to ignore this line. In many cases, such a choice has the sole implication of training the machine learning algorithm without any observation. Let's say that you have a badly formatted dataset and you want to load just all the good lines and ignore the badly formatted ones.

Here is what you can do with the `error_bad_lines` option:

```
Val1,Val2,Val3

0,0,0

1,1,1

2,2,2,2

3,3,3

In: bad_dataset = pd.read_csv('a_loading_example_2.csv',
error_bad_lines=False)
```

```
bad_dataset
Skipping line 4: expected 3 fields, saw 4
Out:
   Val1  Val2  Val3
0     0     0     0
1     1     1     1
2     3     3     3
```

Dealing with big datasets

If the dataset you want to load is too big to fit in the memory, chances are that you are going to use a batch machine learning algorithm, or you may just need a bit of it (let's say that you want to take a peek at the data). Thanks to Python, you actually can load the data in chunks. This operation is also called data streaming since the dataset flows into a DataFrame or some other data structure as a continuous flow. As opposed to all the previous cases, the dataset has been fully loaded into the memory in a standalone step.

With pandas, there are two ways to chunk and load a fie. The first way to do this is by loading the dataset in chunks of the same size; each chunk is a piece of the dataset that contains all the columns and the number of lines set in the function call (the `chunksize` parameter). Note that the output of the `read_csv` function in this case is not a pandas DataFrame but an iterator-like object. In fact, to get the results, you need to iterate that object.

```
In: import pandas as pd
In: iris_chunks = pd.read_csv(iris_filename, header=None,
names=['C1', 'C2', 'C3', 'C4', 'C5'], chunksize=10)
In: for chunk in iris_chunks:
        print chunk.shape
        print chunk
Out:
(10, 5)
    C1   C2   C3   C4           C5
0  5.1  3.5  1.4  0.2  Iris-setosa
1  4.9  3.0  1.4  0.2  Iris-setosa
2  4.7  3.2  1.3  0.2  Iris-setosa
3  4.6  3.1  1.5  0.2  Iris-setosa
4  5.0  3.6  1.4  0.2  Iris-setosa
```

```
5  5.4  3.9  1.7  0.4  Iris-setosa

6  4.6  3.4  1.4  0.3  Iris-setosa

7  5.0  3.4  1.5  0.2  Iris-setosa

8  4.4  2.9  1.4  0.2  Iris-setosa

9  4.9  3.1  1.5  0.1  Iris-setosa
```

There will be 14 other pieces like these, with each of them having the (10, 5) shape. The other method to load a big dataset is by specifically asking for an iterator of it. In this case, you can decide dynamically the length (that is, how many lines) you want for each piece of the pandas DataFrame.

```
In:  iris_iterator = pd.read_csv(iris_filename, header=None,
names=['C1', 'C2', 'C3', 'C4', 'C5'], iterator=True)

In:  print iris_iterator.get_chunk(10).shape
(10, 5)

In:  print iris_iterator.get_chunk(20).shape
(20, 5)

In:  piece = iris_iterator.get_chunk(2)
piece

Out:

  C1  C2  C3  C4  C5

0  4.8  3.1  1.6  0.2  Iris-setosa

1  5.4  3.4  1.5  0.4  Iris-setosa
```

In this example, we first got the iterator. Then, we got a piece of data with 10 lines. We then got 20 further rows, and finally the two rows that are printed at the end.

Besides pandas, you can also use the `csv` package that offers two functions to iterate small chunks of data from files: the `reader` and the `DictReader` functions. Let's start importing the `csv` package:

```
In:import csv
```

The `reader` reads the data from the disks to the Python lists. `DictReader` instead transforms the data into a dictionary. Both functions work by iterating over the rows of the file being read. The `reader` returns exactly what it reads, stripped of the return carriage and split into a list by the separator (which is by default the comma, but this can be modified). `DictReader` will map the list's data into a dictionary whose keys will be defined by the first row (if a header is present) or the parameter fieldnames (using a list of strings that report the column names).

The reading of lists in a native manner is not a limitation. For instance, it will be easier to speed up the code using a fast Python implementation such as PyPy. Moreover, we can always convert lists into NumPy ndarrays (a data structure that we are going to introduce you to soon). By reading the data into JSON-style dictionaries, it will be quite easy to get a DataFrame.

Here is a simple example that uses such functionalities from the csv package.

Let's pretend that our datasets-uci-iris.csv file that was downloaded from mldata.org is a huge file that we cannot fully load in the memory (actually, we remember having seen the file at the beginning of this chapter; it is made up of just 150 examples and the csv lacks a header row).

Therefore, our only choice is to load it in chunks. Let's first conduct an experiment.

```
In:
with open(iris_filename, 'rb') as data_stream:
    for n, row in enumerate(csv.DictReader(data_stream,
             fieldnames = ['sepal_length', 'sepal_width',
    'petal_length', 'petal_width', 'target'],
             dialect='excel')):
        if n== 0:
            print n,row
        else:
            break
Out: 0 {'petal_length': '1.4', 'petal_width': '0.2', 'sepal_width':
'3.5', 'sepal_length': '5.1', 'target': 'Iris-setosa'}
```

What does the preceding code do? First of all, it opens a read-binary connection to the file that aliases it as data_stream. Using the with command assures that the file is closed after the commands placed in the preceding indentation are completely executed.

Then, it iterates (for..in..) and enumerates a csv.DictReader call, which wraps the flow of the data from data_stream. Since we don't have a header row in the file, fieldnames provides information about the fields' names. dialect just specifies that we are calling the standard comma-separated csv (later, we'll provide some hints on how to modify this).

Inside the iteration, if the row being read is just the first, then it is printed. Otherwise, the loop is stopped by a break command. The print command presents us with the row number 0 and a dictionary. You can, therefore, recall every piece of data of the row by just calling the keys with the variables' names.

Similarly, we can make the same code work for the `csv.reader` command, as follows:

```
In: with open(iris_filename, 'rb') as data_stream:
    for n, row in enumerate(csv.reader(data_stream,
dialect='excel')):
        if n==0:
            print row
        else:
            break
Out: ['5.1', '3.5', '1.4', '0.2', 'Iris-setosa']
```

Here, the code is even more straightforward, but the output is simpler, providing a list that contains the row values in a sequence.

At this point, based on this second piece of code, we can create a generator that can be called from a for loop. This retrieves data on the fly from the file in the blocks of the size defined by the batch parameter of the function:

```
In:
def batch_read(filename, batch=5):
    # open the data stream
    with open(filename, 'rb') as data_stream:
        # reset the batch
        batch_output = list()
        # iterate over the file
        for n, row in enumerate(csv.reader(data_stream,
dialect='excel')):
            # if the batch is of the right size
            if n > 0 and n % batch == 0:
                # yield back the batch as an ndarray
                yield(np.array(batch_output))
                # reset the batch and restart
                batch_output = list()
            # otherwise add the row to the batch
            batch_output.append(row)
        # when the loop is over, yield what's left
        yield(np.array(batch_output))
```

Similar to the previous example, the data is drawn out, thanks to the `csv.reader` function wrapped by the `enumerate` function that accompanies the extracted list of data along with the example number (which starts from zero). On the basis of the example number, a batch list is either appended with the data list or returned to the main program using the generative `yield` function. This process is repeated until the entire file is read and returned in batches.

```
In:

import numpy as np

for batch_input in batch_read(iris_filename, batch=3):

    print batch_input

    break

Out:

[['5.1' '3.5' '1.4' '0.2' 'Iris-setosa']

 ['4.9' '3.0' '1.4' '0.2' 'Iris-setosa']

 ['4.7' '3.2' '1.3' '0.2' 'Iris-setosa']]
```

Such a function can provide the basic functionality for learning with stochastic gradient descent as it will be presented in *Chapter 4*, *Machine Learning*, where we will come back to this piece of code and expand the example by introducing some more advanced examples.

Accessing other data formats

So far, we have worked on CSV files only. The pandas library offers similar functionality (and functions) to load MS Excel, HDFS, SQL, JSON, HTML, and Stata datasets. Since they're not used often, the understanding of how one can load and handle them is left to you, who can refer to the verbose documentation available on the website. A basic example on how to load an SQL table is available in the code that accompanies the book.

Finally, pandas DataFrames can be created by merging series or other list-like data. Note that scalars are transformed into lists, as follows:

```
In: import pandas as pd

In:  my_own_dataset = pd.DataFrame({'Col1': range(5), 'Col2':
[1.0]*5, 'Col3': 1.0, 'Col4': 'Hello World!'})
my_own_dataset

Out:

   Col1  Col2  Col3          Col4

0     0     1     1  Hello World!
```

```
1     1     1     1  Hello World!
2     2     1     1  Hello World!
3     3     1     1  Hello World!
4     4     1     1  Hello World!
```

It can easily be said that for each of the columns you want stacked together, you provide their names (as the dictionary key) and values (as the dictionary value for that key). As seen in the preceding example, `Col2` and `Col3` are created in two different ways, but they provide the same resulting column of values. In this way, you can create a pandas DataFrame that contains multiple types of data with a very simple function.

In this process, please ensure that you don't mix lists of different sizes, otherwise an exception will be raised, as shown here:

```
In: my_wrong_own_dataset = pd.DataFrame({'Col1': range(5), 'Col2':
'string', 'Col3': range(2)})
…
ValueError: arrays must all be same length
```

To check the type of data in each column, check the `dtypes` attribute:

```
In: my_own_dataset.dtypes
Col1      int64
Col2    float64
Col3    float64
Col4     object
dtype: object
```

The last method seen in the example is very handy if you wish to see whether a datum is categorical, integer numerical, or floating point, and its precision. In fact, sometimes it is possible to increase the processing speed by rounding up floats to integers and casting double precision to single precision floats or by using only a single type of data. Let's see how you can cast the type in the following example. This example can also be seen as a broad example on how to reassign column data:

```
In:  my_own_dataset['Col1'] = my_own_dataset['Col1'].astype(float)
my_own_dataset.dtypes
Out: Col1    float64
Col2    float64
Col3    float64
Col4     object
dtype: object
```

Data preprocessing

We are now able to import the dataset, even a big, problematic one. Now, we need to learn the basic preprocessing routines in order to make it feasible for the next data science step.

First, if you need to apply a function to a limited section of rows, you can create a **mask**. A mask is a series of Boolean values (that is, True or False) that tells whether the line is selected or not.

For example, let's say we want to select all the lines of the iris dataset that have `sepal length` greater than `6`. We can simply do the following:

```
In: mask_feature = iris['sepal_length'] > 6.0
In: mask_feature
0     False
1     False
...
146     True
147     True
148     True
149    False
```

In the preceding simple example, we can immediately see which observations are `True` and which are not (`False`), and which fit the selection query.

Now, let's see how you can use a selection mask on another example. Let's say, we want to substitute the `Iris-virginica` target label with the `New label` label. We can do this by using the following two lines of code:

```
In: mask_target = iris['target'] == 'Iris-virginica'
In: iris.loc[mask_target, 'target'] = 'New label'
```

You'll see that all occurrences of `Iris-virginica` are now replaced by `New label`. The `.loc()` method is explained in the following . Just think of it as a way to access the data of the matrix with the help of rows-column indexes.

To see the new list of the labels in the target column, we can use the `unique()` method. This method is very handy if you want to initially evaluate the dataset:

```
In: iris['target'].unique()
Out: array(['Iris-setosa', 'Iris-versicolor', 'New label'],
dtype=object)
```

If you want to see some statistics about each feature, you can group each column accordingly (eventually, you can also apply a mask):

```
In: grouped_targets_mean = iris.groupby(['target']).mean()
grouped_targets_mean
Out:
                 sepal_length  sepal_width  petal_length  petal_width
target
Iris-setosa             5.006        3.418         1.464        0.244
Iris-versicolor         5.936        2.770         4.260        1.326
New label               6.588        2.974         5.552        2.026
In: grouped_targets_var = iris.groupby(['target']).var()
grouped_targets_var
Out:
                 sepal_length  sepal_width  petal_length  petal_width
target
Iris-setosa          0.124249     0.145180      0.030106     0.011494
Iris-versicolor      0.266433     0.098469      0.220816     0.039106
New label            0.404343     0.104004      0.304588     0.075433
```

Later, if you need to sort the observations using a function, you can use the `.sort()` method, as follows:

```
In: iris.sort_index(by='sepal_length').head()
Out:
    sepal_length  sepal_width  petal_length  petal_width       target
13           4.3          3.0           1.1          0.1  Iris-setosa
42           4.4          3.2           1.3          0.2  Iris-setosa
38           4.4          3.0           1.3          0.2  Iris-setosa
8            4.4          2.9           1.4          0.2  Iris-setosa
41           4.5          2.3           1.3          0.3  Iris-setosa
```

Finally, if your dataset contains a time series (for example, in the case of a numerical target) and you need to apply a `rolling` operation to it (in the case of noisy data points), you can simply do the following:

```
In: smooth_time_series = pd.rolling_mean(time_series, 5)
```

This can be performed for a rolling average of the values. Alternatively, you can also give the following command:

```
In: median_time_series = pd.rolling_median(time_series, 5)
```

This can be performed for a rolling median of the values. In both of these cases, the window had size five samples.

Data selection

The last topic on pandas that we'll focus on is about data selection. Let's start with an example. We might come across a situation where the dataset contains an index column. How do you properly import it with pandas? And then, can we actively exploit it to make our job simpler?

We will use a very simple dataset that contains an index column (this is just a counter and not a feature). To make the example very generic, let's start the index from 100. So, the index of row number `0` is `100`.

```
n,val1,val2,val3
100,10,10,C
101,10,20,C
102,10,30,B
103,10,40,B
104,10,50,A
```

When trying to load a file the classic way, you'll find yourself in a situation where you have `n` as a feature (or a column). Nothing is practically incorrect, but an index should not be used by mistake as a feature. So it is better to keep it separated. If instead, by chance it is used during the learning phase of your model, you may possibly incur a case of "leakage", which is one of the major sources of error in machine learning.

In fact, if the index is a random number, no harm will be done to your model's efficacy. However, if the index contains progressive, temporal, or even informative elements (for example, certain numeric ranges may be used for positive outcomes, and others for the negative ones), you might incorporate into the model leaked information, that it will be impossible to replicate when using your model on fresh data.

```
In: import pandas as pd
In: dataset = pd.read_csv('a_selection_example_1.csv')
dataset
Out:
     n  val1  val2 val3
0  100    10    10    C
1  101    10    20    C
2  102    10    30    B
3  103    10    40    B
4  104    10    50    A
```

Therefore, while loading such a dataset, we might want to specify that `n` is the index column. Since the index `n` is the first column, we can give the following command:

```
In: dataset = pd.read_csv('a_selection_example_1.csv', index_col=0)
dataset
Out:
     val1  val2 val3
n
100    10    10    C
101    10    20    C
102    10    30    B
103    10    40    B
104    10    50    A
```

Here, the dataset is loaded and the index is correct. Now, to access the value of a cell, there are a few ways. Let's list them one by one.

First, you can simply specify the column and the line (by using its index) you are interested in.

To extract the `val3` of the fifth line (indexed with n=104), you can give the following command:

```
In: dataset['val3'][104]
Out: 'A'
```

Do this operation carefully since it's not a matrix and you might be tempted to first input the row and then the column. Remember that it's actually a pandas DataFrame, and the `[]` operator works first on columns and then on the element of the resulting pandas Series.

To have something similar to the preceding method of accessing data, you can use the `.loc()` method:

```
In: dataset.loc[104, 'val3']
Out: 'A'
```

In this case, you should first specify the index and then the columns you're interested in. In this case, the solution is equivalent to the one provided by the `.ix()` method. The latter works with all kinds of indexes (label or positions), and is more flexible.

> Note that `ix()` has to guess what are you referring to. Therefore, if you don't want to mix labels and positional indexes, `loc` and `iloc` are preferred to create a more structured approach.

```
In: dataset.ix[104, 'val3']
Out: 'A'
In: dataset.ix[104, 2]
Out: 'A'
```

Finally, a full-optimized function that specifies the positions (as in a matrix) is `iloc()`. With it, you must specify the cell by using the row number and column number:

```
In: dataset.iloc[4, 2]
Out: 'A'
```

Finally, the retrieving of sub-matrixes is a very intuitive operation; you simply need to specify the lists of indexes instead of scalars:

```
In: dataset[['val3', 'val2']][0:2]
```

This command is equivalent to the following:

```
In: dataset.loc[range(100, 102), ['val3', 'val2']]
```

This is also equivalent to the following:

```
In: dataset.ix[range(100, 102), ['val3', 'val2']]
```

The following command is identical to the preceding commands:

```
In: dataset.ix[range(100, 102), [2, 1]]
```

and:

```
In: dataset.iloc[range(2), [2,1]]
```

In all the cases, the resulting DataFrame is:

```
Out:
  val3  val2
n
100  C  10
101  C  20
```

Working with categorical and textual data

Typically, you'll find yourself dealing with two main kinds of data: categorical and numerical. Numerical data, such as temperature, amount of money, days of usage, or house number, can be composed of either floating point numbers (like 1.0, -2.3, 99.99, …) or integers (like -3, 9, 0, 1, …). Each value that the data can assume has a direct relation with others since they're comparable. In other words, you can say that a feature with a value of 2.0 is greater (actually, it is double) than a feature that assumes a value of 1.0. This type of data is very well-defined and comprehensible, with binary operators such as equal, greater, and less.

The other type of data you might see in your career is the categorical type. A categorical datum expresses an attribute that cannot be measured and assumes values in a finite or infinite set of values, often named levels. For example, weather is a categorical feature since it takes values in the discrete set [sunny, cloudy, snowy, rainy, and foggy]. Other examples are features that contain URLs, IPs, device brands, items you put in your e-commerce cart, devices IDs, and so on. On this data, you cannot define the equal, greater and less binary operators and therefore, you cannot rank them.

A plus point for both categorical and numerical values are Booleans. In fact, they can be seen as categorical (presence/absence of a feature) or, on the other side, as the probability of a feature having an exhibit (has displayed, has not displayed). Since many machine learning algorithms do not allow the input to be categorical, Boolean features are often used to encode categorical features as numerical values.

Let's continue the example of the weather. If we want to map a feature that contains the current weather and which takes values in the set [`sunny`, `cloudy`, `snowy`, `rainy`, and `foggy`] and encodes them to binary features, we should create five True/False features, with one for each level of the categorical feature. Now, the map is pretty straightforward:

```
Categorical_feature = sunny      binary_features = [1, 0, 0, 0, 0]
Categorical_feature = cloudy     binary_features = [0, 1, 0, 0, 0]
Categorical_feature = snowy      binary_features = [0, 0, 1, 0, 0]
Categorical_feature = rainy      binary_features = [0, 0, 0, 1, 0]
Categorical_feature = foggy      binary_features = [0, 0, 0, 0, 1]
```

Only one binary feature reveals the presence of the categorical feature; the others remain `0`. With this easy step, we moved from the categorical world to a numerical one. The price of this operation is its complexity; instead of a single feature, we now have five features. Generically, instead of a single categorical feature with *N* possible levels, we will create *N* features, each with two numerical values (1/0).
This operation is named dummy coding.

The pandas helps us in this operation, making the mapping easy with a command:

```
In: import pandas as pd

In: categorical_feature = pd.Series(['sunny', 'cloudy', 'snowy',
'rainy', 'foggy'])

mapping = pd.get_dummies(categorical_feature)

mapping

Out:

   cloudy  foggy  rainy  snowy  sunny

0       0      0      0      0      1

1       1      0      0      0      0

2       0      0      0      1      0

3       0      0      1      0      0

4       0      1      0      0      0
```

The output is a DataFrame that contains the categorical levels as column labels and the respective binary features along the column. To map a categorical value to a list of numerical ones, just use the power of pandas:

```
In: mapping['sunny']

Out:

0    1

1    0

2    0

3    0

4    0

In: mapping['cloudy']

Out:

0    0

1    1

2    0

3    0

4    0
```

As seen in this example, `sunny` is mapped into the list of Boolean values [`1`, `0`, `0`, `0`, `0`], `cloudy` to [`0`, `1`, `0`, `0`, `0`], and so on.

The same operation can be done with another toolkit, Scikit-learn. It's somehow more complex since you must first convert text to categorical indices, but the result is the same. Let's take a peek at the previous example again:

```
In: from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
ohe = OneHotEncoder()
levels = ['sunny', 'cloudy', 'snowy', 'rainy', 'foggy']
fit_levs = le.fit_transform(levels)
ohe.fit([[fit_levs[0]], [fit_levs[1]], [fit_levs[2]], [fit_levs[3]],
[fit_levs[4]]])
print ohe.transform([le.transform(['sunny'])]).toarray()
print ohe.transform([le.transform(['cloudy'])]).toarray()
Out: array([[ 0.,  0.,  0.,  0.,  1.]])
array([[ 1.,  0.,  0.,  0.,  0.]])
```

Basically, `LabelEncoder` maps the text to a *0*-to-*N* integer number (Note that in this case, it's still a categorical variable since it makes no sense to rank it). Now, these five values are mapped to five binary variables.

A special type of data – text

Let's introduce another type of data. Text is a frequently used input for machine learning algorithms since it contains a natural representation of data in our language. It's so rich that it also contains the answer to what we're looking for. The most common approach when dealing with text is to use a bag of words. According to this approach, every word becomes a feature and the text becomes a vector that contains non-zero elements for all the features (i.e. the words) in its body. Given a text dataset, what's the number of features? It is simple. Just extract all the unique words in it and enumerate them. For a very rich text that uses all the English words, that number is in the 1 million band. If you're not going to further process it (removal of the 3rd person (s), abbreviations, contractions, and acronyms), you might find yourself dealing with more than that, but that's a very rare case. In a plain and simple approach, which is the target of this book, we just let Python do its best.

The dataset used in this section is textual; it's the famous 20newsgroup (for more information about this, visit `http://qwone.com/~jason/20Newsgroups/`). It is a collection of about twenty thousand documents that belong to 20 topics of newsgroups. It's one of the most frequently used (if not the top most used) datasets while dealing with text classification and clustering. To import it, we're going to use only its restricted subset, which contains all the science topics (medicine and space):

```
In: from sklearn.datasets import fetch_20newsgroups
categories = ['sci.med', 'sci.space']
twenty_sci_news = fetch_20newsgroups(categories=categories)
```

The first time you run this command, it automatically downloads the dataset and places it in the `$HOME/scikit_learn_data/20news_home/` default directory. You can query the dataset object by asking for the location of the files, their content, and the label (that is, the topic of the discussion where the document was posted). They're located in the `.filenames`, `.data`, and `.target` attributes of the object respectively.

```
In: twenty_sci_news.data[0]

Out: From: flb@flb.optiplan.fi ("F.Baube[tm]")
Subject: Vandalizing the sky

X-Added: Forwarded by Space Digest

Organization: [via International Space University]

Original-Sender: isu@VACATION.VENARI.CS.CMU.EDU

Distribution: sci

Lines: 12

From: "Phil G. Fraering" <pgf@srl03.cacs.usl.edu>
[...]

In: twenty_sci_news.filenames

Out: array([
'/Users/datascientist/scikit_learn_data/20news_home/20news-bydate-
train/sci.space/61116',
       '/Users/datascientist/scikit_learn_data/20news_home/20news-
bydate-train/sci.med/58122',
       '/Users/datascientist/scikit_learn_data/20news_home/20news-
bydate-train/sci.med/58903',
       ...,
       '/Users/datascientist/scikit_learn_data/20news_home/20news-
bydate-train/sci.space/60774',
[...]

In: print twenty_sci_news.target[0]

print twenty_sci_news.target_names[twenty_sci_news.target[0]]

Out:

1

sci.space
```

The target is categorical, but it's represented as an integer (`0` for `sci.med` topic and `1` for `sci.space`). If you want to read it out, check against the index of the `twenty_sci_news.target` array.

The easiest way to deal with the text is by transforming the body of the dataset into a series of words. This means that for each document, the number of times a specific word appears in the body will be counted.

For example, let's make a small, easy-to-process dataset:

- `Document_1`: we love data science
- `Document_2`: data science is hard

In the entire dataset, which contains `Document_1` and `Document_2`, there are only six different words: we, `love`, `data`, `science`, `is`, and `hard`. Given this array, we can associate each document to a feature vector:

```
Feature_Document_1 = [1 1 1 1 0 0]
Feature_Document_2 = [0 0 1 1 1 1]
```

Note that we're discarding the position of the words and retaining only the number of times the word appears in the document. That's all.

In the `20newsletter` database, with Python, this can be done in a simple way:

```
In: from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()
word_count = count_vect.fit_transform(twenty_sci_news.data)
word_count.shape
Out: (1187, 25638)
```

First, we instantiate a `CountVectorizer` object. Then, we call the method to count the terms in each document and produce a feature vector for each of them (`fit_transform`). We then query the matrix size. Note that the output matrix is sparse since it's very common to have only a limited selection of words for each document (because the number of nonzero elements in each line is very low and it makes no sense to store all the redundant zeros). Anyway, the output shape is `(1187, 25638)`. The first value is the number of observations in the dataset (the number of documents), while the latter is the number of features (the number of unique words in the dataset).

After the `CountVectorizer` transforms, each document is associated with its feature vector. Let's take a look at the first document:

```
In: print word_count[0]
Out:(0, 10827)  2
  (0, 10501)  2
  (0, 17170)  1
  (0, 10341)  1
  (0, 4762)  2
  (0, 23381)	2
  (0, 22345)	1
  (0, 24461)	1
  (0, 23137)	7
[...]
```

You can see that the output is a sparse vector where only nonzero elements are stored. To see the direct correspondence to the words, try the following code:

```
In: word_list = count_vect.get_feature_names()
for n in word_count[0].indices:
    print "Word:", word_list[n], "appears", word_count[0, n], "times"
Out: Word: from appears 2 times
Word: flb appears 2 times
Word: optiplan appears 1 times
Word: fi appears 1 times
Word: baube appears 2 times
Word: tm appears 2 times
Word: subject appears 1 times
Word: vandalizing appears 1 times
Word: the appears 7 times
[...]
```

So far, everything was pretty simple. Let's move forward to complexity and effectiveness. Counting words is good, but we can do more—compute their frequency. It's a measure that you can compare across differently sized datasets. It gives an idea whether a word is a stop word (that is, a very common word such as a, an, the, and is) or a rare, unique one. Typically, these terms are the most important because they're able to characterize an instance and the features based on these words, which are very discriminative in the learning process. To retrieve the frequency of each word in each document, try the following code:

```
In: from sklearn.feature_extraction.text import TfidfVectorizer
tf_vect = TfidfVectorizer(use_idf=False, norm='l1')
word_freq = tf_vect.fit_transform(twenty_sci_news.data)
word_list = tf_vect.get_feature_names()
for n in word_freq[0].indices:
    print "Word:", word_list[n], "has frequency", word_freq[0, n]
Out: Word: from has frequency 0.021978021978
Word: flb has frequency 0.021978021978
Word: optiplan has frequency 0.010989010989
Word: fi has frequency 0.010989010989
```

```
Word: baube has frequency 0.021978021978
Word: tm has frequency 0.021978021978
Word: subject has frequency 0.010989010989
Word: vandalizing has frequency 0.010989010989
Word: the has frequency 0.0769230769231
[…]
```

The sum of the frequencies is 1 (or close to 1 due to the approximation). This happens because we chose the `l1` norm. In this specific case, the word frequency is a probability distribution function. Sometimes, it's nice to increase the difference between rare and common words. In that case, you can use the `l2` norm to normalize the feature vector.

An even more effective way to vectorize text data is by using Tfidf. In brief, you can multiply the term frequency of the words that compose a document by the inverse document frequency of the word itself (that is, in the number of documents it appears, if logarithmically scaled). This is very handy to highlight words that effectively describe each document, and is a powerful discriminative element among the dataset.

```
In: from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vect = TfidfVectorizer() # Default: use_idf=True
word_tfidf = tfidf_vect.fit_transform(twenty_sci_news.data)
word_list = tfidf_vect.get_feature_names()
for n in word_tfidf[0].indices:
    print "Word:", word_list[n], "has tfidf", word_tfidf[0, n]
Out:
Word: fred has tfidf 0.0893604523484
Word: twilight has tfidf 0.139389277822
Word: evening has tfidf 0.113026734241
Word: in has tfidf 0.0239166759663
Word: presence has tfidf 0.118805671173
Word: its has tfidf 0.0614868335851
Word: blare has tfidf 0.150393472236
Word: freely has tfidf 0.118805671173
Word: may has tfidf 0.0543546855668
Word: caste has tfidf 0.43258869397
Word: baube has tfidf 0.26428200052
Word: flb has tfidf 0.26428200052
```

```
Word: tm has tfidf 0.219051510033
[...]
```

In this example, the four most characterizing words of the first documents are `caste`, `baube`, `flb`, and `tm` (they have the highest `tfidf` score). This means that their term frequency within the document is high, whereas they're pretty rare in the remaining documents.

So far, for each word, we have generated a feature. What about taking a pair of words together? That's exactly what happens when you consider bigrams instead of unigrams. With bigrams (or generically, n-grams), the presence or absence of a word as well as its neighbors matters (that is, the words near it and their disposition). Of course you can mix unigrams and n-grams together and create a rich feature vector for each document. On a trivial example, let's see how n-grams work:

```
In: text_1 = 'we love data science'
text_2 = 'data science is hard'
documents = [text_1, text_2]
documents
```

```
Out: ['we love data science', 'data science is hard']
```

```
In: # That is what we say above, the default one
count_vect_1_grams = CountVectorizer(ngram_range=(1, 1),
stop_words=[], min_df=1)
word_count = count_vect_1_grams.fit_transform(documents)
word_list = count_vect_1_grams.get_feature_names()
print "Word list = ", word_list
print "text_1 is described with", [word_list[n] + "(" +
str(word_count[0, n]) + ")" for n in word_count[0].indices]
```

```
Out: Word list =  [u'data', u'hard', u'is', u'love', u'science',
u'we']
text_1 is described
```

```
 with [u'we(1)', u'love(1)', u'data(1)', u'science(1)']
```

```
In: # Now a bi-gram count vectorizer
count_vect_1_grams = CountVectorizer(ngram_range=(2, 2))
word_count = count_vect_1_grams.fit_transform(documents)
word_list = count_vect_1_grams.get_feature_names()
print "Word list = ", word_list
print "text_1 is described with", [word_list[n] + "(" +
str(word_count[0, n]) + ")" for n in word_count[0].indices]
Out: Word list =  [u'data science', u'is hard', u'love data',
u'science is', u'we love']
text_1 is described with [u'we love(1)', u'love data(1)', u'data
science(1)']
```

```
In: # Now a uni- and bi-gram count vectorizer
count_vect_1_grams = CountVectorizer(ngram_range=(1, 2))
word_count = count_vect_1_grams.fit_transform(documents)
word_list = count_vect_1_grams.get_feature_names()
print "Word list = ", word_list
print "text_1 is described with", [word_list[n] + "(" +
str(word_count[0, n]) + ")" for n in word_count[0].indices]
```

```
Out: Word list =  [u'data', u'data science', u'hard', u'is', u'is
hard', u'love', u'love data', u'science', u'science is', u'we', u'we
love']
text_1 is described with [u'we(1)', u'love(1)', u'data(1)',
u'science(1)', u'we love(1)', u'love data(1)', u'data science(1)']
```

The last one very intuitively composes the first and second approach. In this example, we used a `CountVectorizer`, but this approach is very common with a `TfidfVectorizer`. Note that the number of features explodes exponentially when you use n-grams.

If you have too many features (the dictionary may be too rich, there may be too many n-grams, or the computer may be just limited), you can use a trick that lowers the complexity of the problem (but you should first evaluate the trade-off performance /trade-off complexity). It's common to use the hashing trick where many words (or n-grams) are hashed and their hashes collide (which makes a bucket of words). Buckets are sets of semantically unrelated words, but with colliding hashes. With `HashingVectorizer()`, as shown in the following example, you can decide the number of buckets of words you want. The resulting matrix, of course, reflects your setting:

```
In: from sklearn.feature_extraction.text import HashingVectorizer
hash_vect = HashingVectorizer(n_features=1000)
word_hashed = hash_vect.fit_transform(twenty_sci_news.data)
word_hashed.shape
```

```
Out: (1187, 1000)
```

Note that you can't invert the hashing process (since it's a digest operation). Therefore, after this transformation, you will have to work on the hashed features as they are.

Data processing with NumPy

Having introduced the essential pandas commands to upload and preprocess your data in memory completely, or even in smaller batches (or in single data rows), you'll have to work on it in order to prepare a suitable data matrix for your supervised and unsupervised learning procedures.

As a best practice, we suggest that you divide the task between a phase of your work when your data is still heterogeneous (a mix of numerical and symbolic values) and another phase when it is turned into a numeric table of data arranged in rows that represent your examples, and columns that contain the characteristic observed values of your examples, which are your variables.

In doing so, you'll have to wrangle between two key Python packages for scientific analysis, pandas and NumPy, and their two pivotal data structures, DataFrame and ndarray.

Since the target data structure is a NumPy ndarray object, let's start from the result we want to achieve.

NumPy's n-dimensional array

Python presents native data structures, such as lists and dictionaries that you should use to the best of your ability. Lists, for example, can store sequentially heterogeneous objects (for instance, you can save numbers, texts, images, and sounds in the same list). On the other hand, Dictionaries, on the basis of a lookup table (a hash table), can recall content. The content can be any Python object, and often, it is a list of another dictionary. Thus, Dictionaries allow you to access a complex, multidimensional data structure.

Anyway, lists and dictionaries have their own limitations. First of all, there's the problem with memory and speed. They are not really optimized for using nearly contiguous chunks of memory, and this may become a problem when trying to apply highly optimized algorithms or multiprocessor computations, because the memory handling may turn into a bottleneck. Then, they are excellent for storing data but not for operating on it. So, whatever you may want to do with your data, you have to first define custom functions and iterate or map over the list or dictionary elements. Iterating may often prove suboptimal when working on a large amount of data.

NumPy offers an ndarray object class (n-dimensional array) that has the following attributes:

- It is memory optimal (and, besides other aspects, configured to transmit data to C or Fortran routines in the best performing layout of memory blocks)
- It allows fast linear algebra computations (vectorization) and element-wise operations (broadcasting) without any need to use iterations with for loops
- It is the data structure that critical libraries, such as SciPy or Scikit-learn, expect as an input for their functions

All of this comes with some limitations. In fact, ndarray objects have the following drawbacks:

- They usually store only elements of a single, specific data type that you can define beforehand (but there's a way to define complex data and heterogeneous data types, though they could be very difficult to handle for analysis purposes)
- After they are initialized, their size is fixed

The basics of NumPy ndarray objects

In Python, an array is basically a block of memory-contiguous data of a specific type with a header that contains the indexing scheme and the data type descriptor.

Thanks to the indexing scheme, an array can represent a multidimensional data structure where each element is indexed with a tuple of n integers, where n is the number of dimensions. Therefore, if your array is unidimensional, that is, a vector of sequential data, the index will start from zero (as in Python lists).

If it is bidimensional, you'll have to use two integers as an index (a tuple of coordinates of the type *x,y*); if there are three dimensions, the number of integers used will be three (a tuple *x,y,z*), and so on.

At each indexed location, the array will contain data of the specified data type. There are many numerical data types, strings, and other Python objects that can be stored in an array. It is also possible to create custom data types and therefore handle data sequences of different types, though we advise against it and suggest that in such cases, you should use the pandas DataFrame, which is much more flexible for the intensive usage necessary for a data scientist. Consequently, we will consider only arrays of a specific, defined type.

Since the type (and the memory space it occupies in terms of bytes) of an array should be defined from the beginning, the array creation procedure can reserve the exact memory space to contain all the data. The access, modification, and computation of the elements of an array are therefore quite fast, though this also consequently implies that the array is fixed and cannot be structurally changed.

For our purposes, it is therefore very important to understand that when we are *viewing* an array, we have called a procedure that allows us to immediately convert the data into something else (but the sourcing array has been unaltered), and when we are *copying* an array, we are creating a new array with a different structure (thus occupying new fresh memory).

Creating NumPy arrays

There is more than one way to create NumPy arrays. The following are some of the ways:

- Transforming an existing data structure into an array
- Creating an array from scratch and populating it with default or calculated values
- Uploading some data from a disk into an array

If you are going to transform an existing data structure, the odds are in favor of you working with a structured list or a pandas DataFrame.

From lists to unidimensional arrays

One of the most common situations you will encounter when working with data is the transforming of a list into an array.

When operating such a transformation, it is important to consider the objects the lists contain because this will determine the dimensionality and the dtype of the resulting array.

Let's start with the first example of a list containing just integers:

```
In: import numpy as np
In:  # Transform a list into a uni-dimensional array
list_of_ints = [1,2,3]
Array_1 = np.array(list_of_ints)
In: Array_1
Out: array([1, 2, 3])
```

Remember that you can access a monodimensional array as you do with a standard Python list (the indexing starts from zero):

```
In: Array_1[1] # let's output the second value
Out: 2
```

We can ask for further information about the type of the object and the type of its elements (the effectively resulting type depends on whether your system is 32-bit or 64-bit):

```
In: type(Array_1)
Out: numpy.ndarray
In: Array_1.dtype
Out: dtype('int32')
```

> The default dtype depends on the system you're operating.

Our simple list of integers will turn into a unidimensional array, that is, a vector of 32-bit integers (ranging from -231 to 231-1, the default integer on the platform we used for our examples).

Controlling the memory size

You may think that it is a waste of memory using an int32 if the range of your values is so limited.

In fact, conscious of data-intensive situations, you can calculate how much memory space your Array_1 object is taking:

```
In: import numpy as np
In: Array_1.nbytes
Out: 12
```

> Please note that on 64bit platforms the result will be 24.

In order to save memory, you can specify beforehand the type that best suits your array:

```
In: Array_1 = np.array(list_of_ints, dtype= 'int8')
```

Now, your simple array occupies just a fourth of the previous memory space. It may seem an obvious and overly simplistic example, but when dealing with millions of examples, defining the best data type for your analysis can really save the day, allowing you to fit everything in memory.

For your reference, here are a few tables that present the most common data types for data science applications and their memory usage for a single element:

Type	Size in bytes	Description
bool	1	Boolean (True or False) stored as a byte
int_	4	Default integer type (normally int32 or int64)
int8	1	Byte (-128 to 127)
int16	2	Integer (-32768 to 32767)
int32	4	Integer (-2**31 to 2**31-1)

Type	Size in bytes	Description
int64	8	Integer (-2**63 to 2**63-1)
uint8	1	Unsigned integer (0 to 255)
uint16	2	Unsigned integer (0 to 65535)
uint32	3	Unsigned integer (0 to 2**32-1)
uint64	4	Unsigned integer (0 to 2**64-1)
float_	8	Shorthand for float64
float16	2	Half-precision float (exponent 5 bits, mantissa 10 bits)
float32	4	Single-precision float (exponent 8 bits, mantissa 23 bits)
float64	8	Double-precision float (exponent 11 bits, mantissa 52 bits)

> There are some more numerical types, such as the complex number which are less usual but which may be required by your application (for example, in a spectrogram). You can get the full picture from the NumPy user guide at `http://docs.scipy.org/doc/numpy/user/basics.types.html`.

If an array has a type that you want to change, you can easily create a new array by casting a new specified type:

```
In: Array_1b = Array_1.astype('float32')
Array_1b
Out: array([ 1.,  2.,  3.], dtype=float32)
```

In case your array is quite memory consuming, note that the `.astype` method will always create a new array.

Heterogeneous lists

What if the list were made of heterogeneous elements, such as integers, floats, and strings?

This gets more tricky. A quick example can describe the situation to you:

```
In: import numpy as np
In: complex_list = [1,2,3] + [1.,2.,3.] + ['a','b','c']
In: Array_2 = np.array(complex_list[:3]) # at first the input list is
just ints
print 'complex_list[:3]', Array_2.dtype
```

```
Array_2 = np.array(complex_list[:6]) # then it is ints and floats
print 'complex_list[:6]', Array_2.dtype
Array_2 = np.array(complex_list)     # finally we add strings
print 'complex_list[:] ',Array_2.dtype
Out:
complex_list[:3] int32
complex_list[:6] float64
complex_list[:]  |S3
```

As explicated by our output, it seems that float types prevail over int types and strings (`|S3` which means string of size three or less) prevail over everything else.

While creating an array using lists, you can mix different elements, and the most pythonic way to check the results is by actually questioning the `dtype` of the resulting array.

Be aware that if you are uncertain about the contents of your array, you really have to check. Otherwise, you may later find it impossible to operate certain operations on your resulting array and incur an error (unsupported operand type):

```
In: # Check if a NumPy array is of the desired numeric type
print isinstance(Array_2[0],np.number)
Out: False
```

In our data munging process, unintentionally finding out an array of the string type as output would mean that we forgot to transform all variables into numeric ones in the previous steps—for example, when all the data was stored in a pandas DataFrame. In the previous paragraph, *Working with textual data*, we provided some simple and straightforward ways to deal with such situations.

Before that, let's complete our overview of how to derive an array from a list object. As we mentioned before, the type of objects in the list influences the dimensionality of the array, too.

From lists to multidimensional arrays

If a list containing numeric or textual objects is rendered into a unidimensional array (that could represent a coefficient vector, for instance), a list of lists translates into a bidimensional array and a list of list of lists becomes a three-dimensional one.

```
In: import numpy as np
In: # Transform a list into a bidimensional array
a_list_of_lists = [[1,2,3],[4,5,6],[7,8,9]]
```

```
Array_2D = np.array(a_list_of_lists )
Array_2D
Out: array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

As mentioned before, you can call out single values with indices, as in a list, though here you'll have two indices—one for the row dimension (also called axis 0) and one for the column dimension (axis 1).

```
In: Array_2D[1,1]
Out: 5
```

Two-dimensional arrays are usually the norm in data science problems, though a third dimension (a discrete time dimension, for instance) may sometimes be found:

```
In: # Transform a list into a multi-dimensional array
a_list_of_lists_of_lists = [[[1,2],[3,4],[5,6]],
[[7,8],[9,10],[11,12]]]
Array_3D = np.array(a_list_of_lists_of_lists)
Array_3D
Out: array([[[ 1,  2],
        [ 3,  4],
        [ 5,  6]],
       [[ 7,  8],
        [ 9, 10],
        [11, 12]]])
```

To access single elements of a three-dimensional array, you just have to point out a tuple of three indexes:

```
In: Array_3D[0,2,0] # Accessing the 5th element
Out: 5
```

Arrays can be made from tuples in a way that is similar to the method of creating lists. Also, dictionaries can be turned into two-dimensional arrays thanks to the `.items()` method, which returns a copy of the dictionary's list of key and value pairs.

```
In: np.array({1:2,3:4,5:6}.items())
Out: array([[1, 2],
[3, 4],
[5, 6]])
```

Resizing arrays

Earlier, we mentioned about how you can change the type to the elements of an array. We will now shortly stop awhile to examine the most common instructions to modify the shape of an existing array.

Let's start with an example that uses the `.reshape` method, which accepts as parameter an n-tuple containing the size of the new dimensions:

```
In: import numpy as np
In: # Restructuring a NumPy array shape
original_array = np.array([1, 2, 3, 4, 5, 6, 7, 8])
Array_a = original_array.reshape(4,2)
Array_b = original_array.reshape(4,2).copy()
Array_c = original_array.reshape(2,2,2)
# Attention because reshape creates just views, not copies
original_array[0] = -1
```

Our original array is a unidimensional vector of integer numbers from one to eight.

- We assign `Array_a` to a reshaped `original_array` of size 4x2
- We do the same with `Array_b`, though we append the `.copy()` method that will copy the array into a new one
- Finally we assign `Array_c` to a reshaped array in three dimensions of size 2x2x2
- After having done such assignment, the first element of `original_array` is changed in value from `1` to `-1`

Now, if we check the content of our arrays, we will notice that `Array_a` and `Array_c`, though they have the desired shape, are characterized by a -1 first element. That's because they dynamically mirror the original array they are a view from.

```
In: Array_a
Out: array([[-1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
In: Array_c
Out: array([[[-1,  2],
  [3,  4]],
  [[ 5,  6],
  [7,  8]]])
```

Only the `Array_b` array, having been copied before mutating the original array, has a first element with a value of `1`.

```
In: Array_b
Out: array([[1, 2],
   [3, 4],
  [5, 6],
  [7, 8]])
```

If it is necessary to change the shape of the original array, then the `resize` method is to be favored.

```
In: original_array.resize(4,2)
   original_array
Out: array([[-1,  2],
   [ 3,  4],
  [ 5,  6],
   [ 7,  8]])
```

Basically, the same results may be obtained acting on the `.shape` value by assigning a tuple of values representing the size of the intended dimensions.

```
In: original_array.shape = (4,2)
```

Instead, if your array is bidimensional and you need to exchange the rows with the columns, that is, to transpose the array, the `.T` or `.transpose()` methods will help you obtain this kind of transformation (which is a view like `.reshape`).

```
In: original_array
Out:
array([[-1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8]])
```

Arrays derived from NumPy functions

If you need a vector or a matrix characterized by particular numeric series (zeros, ones, ordinal numbers, and particular statistical distributions), NumPy functions provide you with quite a large range of choices.

First of all, creating a NumPy array of ordinal values (integers) is straightforward if you use the `arange` function, which returns integer values in a given interval (usually from zero) and reshapes its results:

```
In: import numpy as np
In: ordinal_values = np.arange(9).reshape(3,3)
ordinal_values
Out: array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

If the array has to be reversed in the order of values, give the following command:

```
In: np.arange(9)[::-1]
```

If the integers are just random (with no order, and possibly repeated), use the following command:

```
In: np.random.randint(low=1,high=10,size=(3,3)).reshape(3,3)
```

Other useful arrays are either made of just zeros and ones, or are identity matrices:

```
In: np.zeros((3,3))
In: np.ones((3,3))
In: np.eye(3)
```

If the array will be used for a grid search to search optimal parameters, fractional values in an interval or a logarithmic growth should prove most useful:

```
In: fractions = np.linspace(start=0, stop=1, num=10)
fractions
In: growth = np.logspace(start=0, stop=1, num=10, base=10.0)
```

Instead, statistical distributions such as normal or uniform may be useful for the initialization of a vector or matrix of coefficients.

A 3x3 matrix of standardized normal values (mean=0, std=1):

```
In: std_gaussian = np.random.normal(size=(3,3))
```

If you need to specify a different mean and standard deviation, give the following command:

```
In: gaussian = np.random.normal(loc=1.0, scale= 3.0, size=(3,3))
```

The loc parameter stands for the mean and the scale is actually the standard deviation.

Another frequent choice for a statistical distribution that is used to initialize a vector is certainly the uniform distribution:

```
In: np.random.uniform(low=0.0, high=1.0, size=(3,3))
```

Getting an array directly from a file

NumPy arrays can also be created directly from the data present in a file.

Let's use an example from the previous chapter:

```
In: import numpy as np
In: housing = np.loadtxt('regression-datasets-
housing.csv',delimiter=',', dtype=float)
```

NumPy loadtxt, given a filename, delimiter and a dtype, will upload the data to an array, unless the dtype is wrong. For instance, there's a string variable and the required array type is a float, as shown in the following example:

```
In: np.loadtxt('datasets-uci-iris.csv',delimiter=',',dtype=float)
Out: ValueError: could not convert string to float: Iris-setosa
```

Extracting data from pandas

Interacting with pandas is quite easy. In fact, with pandas being built upon NumPy, arrays can be easily extracted from DataFrame objects, and they can be transformed into a DataFrame themselves.

First of all, let's upload some data into a DataFrame. The BostonHouse example we downloaded in the previous chapter from the ML repository is perfect:

```
In: import pandas as pd
In: import numpy as np
In: housing_filename = 'regression-datasets-housing.csv'
housing = pd.read_csv(housing_filename,  header=None)
```

At this point, the .values method will extract an array of a type that tries to accommodate all the different types present in the DataFrame, as demonstrated in the previous section, *Heterogeneous lists*.

```
In: housing_array = housing.values
housing_array.dtype
Out: dtype('float64')
```

In such a case, the selected type is float64 because the float type prevails over int types:

```
In: housing.dtypes

Out: 0     float64

1       int64

2     float64

3       int64

4     float64

5     float64

6     float64

7     float64

8       int64

9       int64

10      int64

11    float64

12    float64

13    float64

dtype: object
```

Asking for the types used by the DataFrame object before extracting your NumPy array by using the `.dtypes` method on the DataFrame allows you to anticipate the `dtype` of the resulting array and consequently decide whether to transform or change the type of the variables in the DataFrame object before proceeding (please see the section, *Working with categorical and textual data* of this chapter).

NumPy fast operation and computations

When arrays need to be manipulated by mathematical operations, you just need to apply the operation on the array with respect to a numerical constant (a scalar) or an array of the exact same shape:

```
In: import numpy as np

In: a =  np.arange(5).reshape(1,5)

In: a += 1

In: a*a

Out: array([[ 1,  4,  9, 16, 25]])
```

The result will be that the operation will be performed element-wise, that is, every element of the array is operated by either the scalar value or the corresponding element of the other array.

When operating on arrays of different dimensions, it is still possible to obtain element-wise operations without having to restructure the data in case one of the corresponding dimensions is 1. In fact, in such a case, the dimension of size 1 is stretched until it matches the dimension of the corresponding array. This conversion is called *broadcasting*.

For instance:

```
In: a = np.arange(5).reshape(1,5) + 1
b = np.arange(5).reshape(5,1) + 1
a * b
Out: array([[ 1,  2,  3,  4,  5],

[ 2,  4,  6,  8, 10],

[ 3,  6,  9, 12, 15],

[ 4,  8, 12, 16, 20],

[ 5, 10, 15, 20, 25]])
```

The preceding code is equivalent to the following:

```
In: a2 = np.array([1,2,3,4,5] * 5).reshape(5,5)
b2 = a2.T
a2 * b2
```

However, it won't require an expansion of memory of the original arrays in order to obtain pair-wise multiplication.

There furthermore exists a wide range of NumPy functions that can operate element-wise on arrays: `abs()`, `sign()`, `round()`, `floor()`, `sqrt()`, `log()`, and `exp()`.

Other usual operations that could be operated by NumPy functions are `sum()` and `prod()`, which provide the summation and product of the array rows or columns on the basis of the specified axis:

```
In: print a2
Out: [[1 2 3 4 5]
 [1 2 3 4 5]
 [1 2 3 4 5]
 [1 2 3 4 5]
 [1 2 3 4 5]]
In: np.sum(a2, axis=0)
Out: array([ 5, 10, 15, 20, 25])
In: np.sum(a2, axis=1)
Out: array([15, 15, 15, 15, 15])
```

When operating on your data, remember that operations and NumPy functions on arrays are extremely fast compared to simple Python lists. Let's try out a couple of experiments. First, let's try to compare a list comprehension to an array when dealing with a sum of a constant:

```
In: %timeit -n 1 -r 3 [i+1.0 for i in range(10**6)]
%timeit -n 1 -r 3 np.arange(10**6)+1.0
Out: 1 loops, best of 3: 158 ms per loop
1 loops, best of 3: 6.64 ms per loop
```

On IPython, `%time` allows you to easily benchmark operations. The `-n 1` parameter just requires the benchmark to execute the code snippet for only one loop; `-r 3` requires you to retry the execution of the loops (in this case, just one loop) three times and report the best performance recorded from such repetitions.

Results on your computer may vary depending on your configuration and operating system. Anyway, the difference between the standard Python operations and the NumPy ones will remain quite large. Though unnoticeable when working on small datasets, this difference can really impact your analysis when dealing with larger data or looping over and over the same analysis pipeline for parameter or variable selection.

This also happens when applying sophisticated operations, such as finding a square root:

```
In: import math
%timeit -n 1 -r 3 [math.sqrt(i) for i in range(10**6)]
Out:
1 loops, best of 3: 222 ms per loop


In: %timeit -n 1 -r 3 np.sqrt(np.arange(10**6))
Out: 1 loops, best of 3: 6.9 ms per loop
```

Matrix operations

For multiplications apart from element-wise calculations using the `np.dot()` function, you can also apply to your bidimensional arrays matrix calculations such as vector-matrix and matrix-matrix multiplications.

```
In: import numpy as np
M = np.arange(5*5, dtype=float).reshape(5,5)
M
Out: array([[  0.,   1.,   2.,   3.,   4.],
[  5.,   6.,   7.,   8.,   9.],
[ 10.,  11.,  12.,  13.,  14.],
[ 15.,  16.,  17.,  18.,  19.],
[ 20.,  21.,  22.,  23.,  24.]])
```

As an example, we create a 5*5 bidimensional array of ordinal numbers from 0 to 24.

We will then define a vector of coefficients and an array column stacking the vector and its reverse:

```
In: coefs = np.array([1., 0.5, 0.5, 0.5, 0.5])
coefs_matrix = np.column_stack((coefs,coefs[::-1]))
print coefs_matrix
Out:
[[ 1.   0.5]
 [ 0.5  0.5]
 [ 0.5  0.5]
 [ 0.5  0.5]
 [ 0.5  1. ]]
```

We can now multiply the array with the vector using the `np.dot` function:

```
In: np.dot(M,coefs)
Out: array([  5.,  20.,  35.,  50.,  65.])
```

or the vector by the array:

```
In: np.dot(coefs,M)
Out: array([ 25.,  28.,  31.,  34.,  37.])
```

or the array by the stacked coefficient vectors (which is a 5*2 matrix):

```
In: np.dot(M,coefs_matrix)
Out: array([[  5.,   7.],
[ 20.,  22.],
[ 35.,  37.],
[ 50.,  52.],
[ 65.,  67.]])
```

NumPy also offers an object class, matrix, which is actually a subclass of ndarray, inheriting all its attributes and methods. NumPy matrices are exclusively bidimensional (as arrays are actually multi-dimensional) by default. When multiplied, they apply matrix products, not element-wise ones (the same happens when raising powers) and they have some special matrix methods (.H for the conjugate transpose and .I for the inverse). Apart from the convenience of operating in a fashion that is similar to that of MATLAB, they do not offer any other advantage. You may risk confusion in your scripts since you'll have to handle different product notations for matrix objects and arrays.

Slicing and indexing with NumPy arrays

Indexing allows us to take a view of an ndarray by pointing out either what slice of columns and rows to visualize, or an index.

Let's define a working array:

```
In: import numpy as np
In: M = np.arange(10*10, dtype=int).reshape(10,10)
```

Our array is a 10x10 bidimensional array. We can initially start by slicing it to a single dimension. The notation for a single dimension is the same as that in Python lists:

```
[start_index_included:end_index_exclude:steps]
```

Let's say that we want to extract even rows from 2 to 8:

```
In: M[2:9:2,:]
Out: array([[20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
[60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
[80, 81, 82, 83, 84, 85, 86, 87, 88, 89]])
```

After slicing the rows, we can furthermore slice columns by taking only the columns from the index 5:

```
In: M[2:9:2,5:]
Out: array([[25, 26, 27, 28, 29],
[45, 46, 47, 48, 49],
[65, 66, 67, 68, 69],
[85, 86, 87, 88, 89]])
```

As in lists, it is possible to use negative index values in order to start counting from the end. Moreover, a negative number for parameters such as steps reverses the order of the output array, like in the following example where the counting starts from the column index `5`, but in the reverse order, it goes towards the index 0:

```
In: M[2:9:2,5::-1]
Out: array([[25, 24, 23, 22, 21, 20],
[45, 44, 43, 42, 41, 40],
[65, 64, 63, 62, 61, 60],
[85, 84, 83, 82, 81, 80]])
```

We also can create Boolean indexes that point out which rows and columns to select. Therefore, we can replicate the previous example by using a `row_index` and a `col_index` variable:

```
In: row_index = (M[:,0]>=20) & (M[:,0]<=80)
col_index = M[0,:]>=5
M[row_index,:][:,col_index]
Out:array([[20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
[60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
[80, 81, 82, 83, 84, 85, 86, 87, 88, 89]])
```

We cannot contextually use Boolean indexes on both columns and rows in the same square brackets, though we can apply the usual indexing to the other dimension using integer indexes. Consequently, we have to first operate a Boolean selection on rows and then reopen the square brackets and operate a second selection on the first, this time focusing on the columns.

If we need a global selection of elements in the array, we can also use a mask of Boolean values, as follows:

```
In: mask = (M>=20) & (M<=90) & ((M / 10.) % 1 >= 0.5)
M[mask]
Out: array([25, 26, 27, 28, 29, 35, 36, 37, 38, 39, 45, 46, 47, 48,
49, 55, 56, 57, 58, 59, 65, 66, 67, 68, 69, 75, 76, 77, 78, 79, 85,
86, 87, 88, 89])
```

This approach is particularly useful if you need to operate on the partition of the array selected by the mask (for example, `M[mask]=0`).

Another way to point out which elements need to be selected from your array is by providing a row or column index consisting of integers. Such indexes may be defined either by a `np.where()` function that transforms a Boolean condition on an array into indexes, or by simply providing a sequence of integer indexes, where integers may be in a particular order or might even be repeated. Such an approach is called *fancy indexing*.

```
In: row_index = [1,1,2,7]
col_index = [0,2,4,8]
```

Having defined the indexes of your rows and columns, you have to apply them contextually to select elements whose coordinates are given by the tuple of values of both the indexes:

```
In: M[row_index,col_index]
Out: array([10, 12, 24, 78])
```

In this way, the selection will report the following points: (`1,0`),(`1,2`),(`2,4`), and (`7,8`). Otherwise, as seen before, you just have to first select the rows and then the columns, which are separated by square brackets:

```
In: M[row_index,:][:,col_index]
Out: array([[10, 12, 14, 18], [10, 12, 14, 18], [20, 22, 24, 28],
[70, 72, 74, 78]])
```

Finally, please remember that slicing and indexing are just mere views of the data. If you need to create new data from such views, you have to use the `.copy` method on the slice and assign it to another variable. Otherwise, any modification to the original array will be reflected on your slice and vice versa. Copy method is shown in the following.

```
In: N = M[2:9:2,5:].copy()
```

Stacking NumPy arrays

When operating with two-dimensional data arrays, there are some common operations, such as the adding of data and variables, that NumPy functions can render easily and quickly.

The most common such operation is the addition of more cases to your array.

Let's create an array to start off with:

```
In: import numpy as np
```

```
In: dataset = np.arange(10*5).reshape(10,5)
```

Now, let's add a single row and a bunch of rows to be concatenated after each other:

```
In: single_line = np.arange(1*5).reshape(1,5)
```

```
a_few_lines = np.arange(3*5).reshape(3,5)
```

We can first try to add a single line:

```
In: np.vstack((dataset,single_line))
```

All you have to do is to provide a tuple containing the vertical array preceding it and the one following it. In our example, the same command can work if you have more lines to be added:

```
In: np.vstack((dataset,a_few_lines))
```

Or, if you want to add the same single line more than once, the tuple can represent the sequential structure of your newly concatenated array:

```
In: np.vstack((dataset,single_line,single_line))
```

Another common situation is when you have to add a new variable to an existing array. In this case, you have to use `hstack` (h stands for horizontal) instead of the just-presented `vstack` command (where v is vertical).

Let's pretend that you have to add a `bias` of unit values to your original array:

```
In: bias = np.ones(10).reshape(10,1)
np.hstack((dataset,bias))
```

Without reshaping `bias` (this, therefore, can be any data sequence of the same length as the rows of the array), you can add it as a sequence using the `column_stack()` function, which obtains the same result but with fewer concerns regarding data reshaping:

```
In: bias = np.ones(10)
np.column_stack((dataset,bias))
```

Adding rows and columns to two-dimensional arrays is basically all that you need to effectively wrangle your data in data science projects. Now, let's see a couple of more specific functions for slightly different data problems.

First of all, though bidimensional arrays are the norm, you can also operate on a three-dimensional data structure. So `dstack()`, which is analogous to `hstack()` and `vstack()`, but which operates on the third axis, will come quite handy:

```
In: np.dstack((dataset*1,dataset*2,dataset*3))
```

In this example, the third dimension offers the original 2D array with a multiplicand, presenting a progressive rate of change (a time or change dimension).

A further problematic variation could be the insertion of a row or, more frequently, a column to a specific position into your array. As you may recall, arrays are contiguous chunks of memory. Insertion actually requires the recreation of a new array, splitting the original array. The NumPy `insert` command helps you to do so in a fast and hassle-free way:

```
In:  np.insert(dataset, 3, bias, axis=1)
```

You just have to define the array where you wish to insert (`dataset`), the position (index `3`), the sequence you want to insert (in this case, the array `bias`), and the axis along which you would like to operate the insertion (axis `1` is the vertical axis).

Naturally, you can insert entire arrays, not just vectors, such as bias, by ensuring that the array to be inserted is aligned with the dimension along which we are operating the insertion. In this example, in order to insert the same array into itself, we have to transpose it as an inserted element:

```
In: np.insert(dataset, 3, dataset.T, axis=1)
```

You can also make insertions on different axes (in the following case, axis `0`, which is the horizontal one, but you can also operate on any dimension of an array that you may have):

```
In: np.insert(dataset, 3, np.ones(5), axis=0)
```

Basically, what is being done is the original array is split at the specified position along the chosen axis. Then, the split data is concatenated with the new data to be inserted.

Summary

In this chapter, we discussed how pandas and NumPy can provide you with all the tools to load and effectively munge your data.

We started with pandas and its data structures, DataFrames and Series, and conducted you through to the final NumPy bidimensional array, a data structure suitable for subsequent experimentation and machine learning. In doing so, we touched upon subjects such as the manipulation of vectors and matrices, categorical data encoding, textual data processing, fixing missing data and errors, slicing and dicing, merging, and stacking.

The pandas and NumPy surely offer many more functions than the essential building blocks we presented here—the commands and procedures illustrated. You can now take any available raw data and apply all the cleaning and shaping transformations necessary for your data science project.

In the next chapter, we will take our data operations to the next step. In this chapter, we together overviewed all the essential data munging operations necessary for a machine learning process to work. In the next chapter, we will discuss all the operations that can potentially improve or even boost your results.

3
The Data Science Pipeline

Until now, we explored how to load data into Python and process it up to a point to create a dataset as a bidimensional NumPy array of numeric values. At this point, we are ready to get fully immersed into data science and extract meaning from data and potential data products. This chapter and the next chapter on machine learning are the most challenging sections of the entire book.

In this chapter, you will learn how to:

- Briefly explore data and create new features
- Reduce the dimensionality of data
- Spot and treat outliers
- Decide on the score or loss metrics that are the best for your project
- Apply the scientific methodology and effectively test the performance of your machine learning hypothesis
- Select the best feature set
- Optimize your learning parameters

Introducing EDA

Exploratory Data Analysis (**EDA**) is the first step in the data science process. This term was coined by John Tukey in 1977, which was when he first wrote a book emphasizing the importance of Exploratory Data Analysis. It's required to understand the dataset better, check its features and shape, validate some hypothesis that you have in mind, and have a preliminary idea about the next step that you want to pursue in the following data science tasks.

In this section, you will work on the iris dataset, which was already used in the previous chapter. First, let's load the dataset:

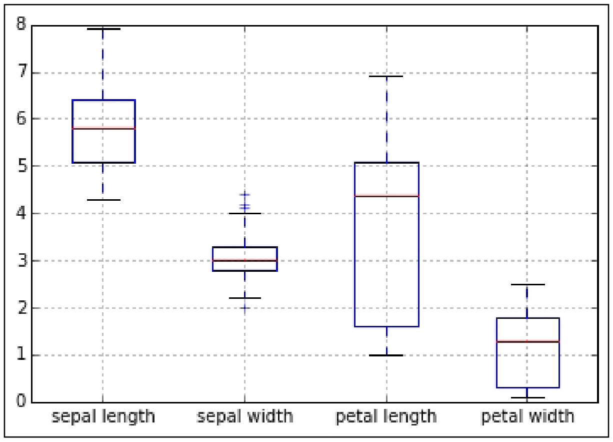
```
In: iris_filename = './datasets-uci-iris.csv'
In: iris = pd.read_csv(iris_filename, header=None, names=
['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'target'])
iris.head()
Out:
   sepal_length  sepal_width  petal_length  petal_width       target
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
```

Great! You have loaded the dataset. Now, the investigation phase starts. Some great insights are provided by the `.describe()` method, which can be used as follows:

```
In: iris.describe()
Out:
  sepal_length  sepal_width  petal_length  petal_width
count  150.000000  150.000000  150.000000  150.000000
mean  5.843333  3.054000  3.758667  1.198667
std  0.828066  0.433594  1.764420  0.763161
min  4.300000  2.000000  1.000000  0.100000
25%  5.100000  2.800000  1.600000  0.300000
50%  5.800000  3.000000  4.350000  1.300000
75%  6.400000  3.300000  5.100000  1.800000
max  7.900000  4.400000  6.900000  2.500000
```

For all numerical features, you have the number of observations, their respective average value, standard deviation, minimum and maximum values, and some percentiles (25 percent, 50 percent, and 75 percent). This gives you a good idea about the distribution of each feature. If you want to visualize this information, just use the `boxplot()` method, as follows:

```
In: iris.boxplot()
Out:
```



Figures in this chapter can be different to the ones obtained on your local computer because graphical layout initialization is made with random parameters.

If you need other quantile values, you can use the `.describe()` method. For example, if you need the 10 percent and the 90 percent, you can try out the following code:

```
In: iris.quantile([0.1, 0.9])
Out:
     sepal_length  sepal_width  petal_length  petal_width
0.1           4.8         2.50           1.4          0.2
0.9           6.9         3.61           5.8          2.2
```

Finally, to calculate the median, you can use the `.median()` method. Similarly, to obtain the mean and standard deviation, the `.mean()` and `.std()`methods are used respectively. In case of categorical features, to get information about the levels (that is, the different values the feature assumes), you can use the `.unique()` method, as follows:

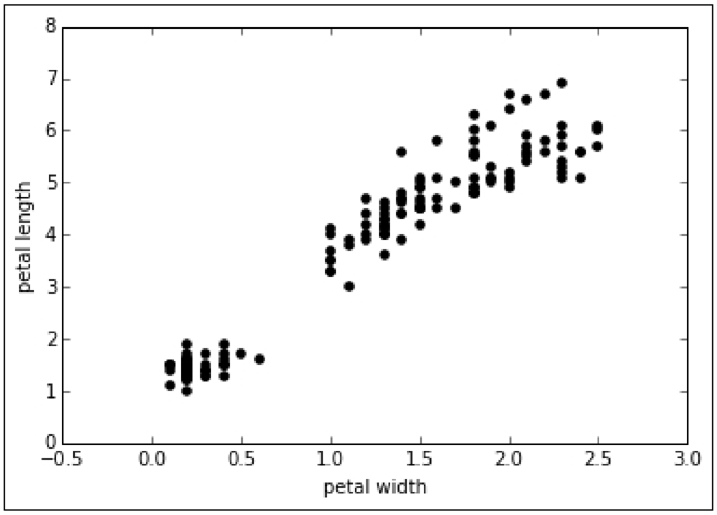
```
In: iris.target.unique()
Out: array(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'],

dtype=object)
```

To examine the relation between features, you can create a co-occurrence matrix. In the following example, we will count the number of times the `petal_length` feature appears more than the average against the same count for the `petal_width` feature. To do this, you need to use the `crosstab` method, as follows:

```
In: pd.crosstab(iris['petal_length'] >
iris['petal_length'].mean(), iris['petal_width'] >
iris['petal_width'].mean())
Out:
petal_width   False  True
petal_length
False            56     1
True              4    89
```

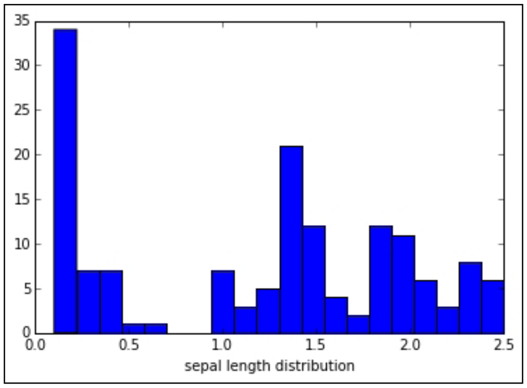
As a result, they will almost always occur conjointly. Therefore, you can suppose a strong relationship between the two events. Graphically, you can take a look at the same behavior by using the following code:

```
In: import matplotlib.pyplot as plt
plt.scatter(iris['petal_width'], iris['petal_length'], alpha=1.0,
color='k')
plt.xlabel("petal width")
plt.ylabel("petal length")
Out:
```



The trend is quite marked; we deduce that *x* and *y* are strongly related. The last operation that you usually perform during an EDA is the checking of the distribution of the feature. To manage this, you can approximate the distribution using a histogram, which can be done with the help of the following snippet:

```
In: plt.hist(iris['petal_width'], bins=20)
plt.xlabel("petal width distribution")
Out:
```



We chose 20 bins after a careful search. In other experiments, 20 might be an extremely low or high value. As a rule of thumb, the starting value is the square root of the number of observations. You will then need to modify it until you recognize a well-known shape of the distribution.

We suggest that you explore such possibilities in all features in order to check their relations and estimate their distribution. In fact, given the distribution, you may decide to treat each feature differently to subsequently achieve the maximum classification or regression performance.

Feature creation

Sometimes, you'll find yourself in a situation where features and target variables are not really related. In this case, you can modify the input dataset, apply linear or nonlinear transformations that can improve the accuracy of the system, and so on. It's a very important step of the process because it completely depends on the skills of the data scientist, who is the one responsible for artificially changing the dataset and shaping the input data for a better fit with the classification model.

For example, if you're trying to predict the value of a house and you just know the height, width, and the length of each room, you can artificially build a feature that represents the volume of the house. This is strictly not an observed feature, but it's a feature built on the top of the existing ones. Let's start with some code:

```
In: import numpy as np
from sklearn import datasets
from sklearn.cross_validation import train_test_split
from sklearn.metrics import mean_squared_error
cali = datasets.california_housing.fetch_california_housing()
X = cali['data']
y = cali['target']
X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.8)
```

We imported the dataset containing the house prices in California. This is a regression problem because the target variable is the house's price (so, a real number). Applying a simple regressor straightaway, the KNN Regressor (there is an in-depth illustration of regressors in *Chapter 4*, *Machine Learning*) ends with a **Mean Absolute Error** (**MAE**) of around 1.15 on the test dataset. Don't worry if you cannot fully understand the code; MAE and the regressors are described later on in the book. Right now, assume that MAE represents an error. So, the lower the value of MAE, the better the solution.

```
In: from sklearn.neighbors import KNeighborsRegressor
regressor = KNeighborsRegressor()
regressor.fit(X_train, y_train)
y_est = regressor.predict(X_test)
print "MAE=", mean_squared_error(y_test, y_est)
Out: MAE= 1.14943289323
```

Now, a result of 1.15 is good, but let's strive to do better. We're going to normalize the input features using Z-scores and compare the regression tasks on this new feature set. Z-normalization is simply the mapping of each feature to a new one with a null mean and unitary variance. With sklearn, this is achieved in the following way:

```
In: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
regressor = KNeighborsRegressor()
regressor.fit(X_train_scaled, y_train)
```

```
y_est = regressor.predict(X_test_scaled)
print "MAE=", mean_squared_error(y_test, y_est)
Out: MAE= 0.412460391361
```

With the help of this easy step, we drop the MAE by more than a half, which now has a value of 0.41.

> Note that we didn't use the original features; we used their linear modification.

Now, let's try to add a nonlinear modification to a specific feature. We can assume that the output is roughly related to the number of occupiers of a house. In fact, there is a big difference between the price of a house occupied by a single person and the price for three persons staying in the same house. However, the difference—between the price for the same for 10 people living there and the price for 12 people—is not that great (though there is still a difference of two). So, let's try to add another feature built as a nonlinear transform of another one:

```
In: non_linear_feat = 5 # AveOccup
# Creating new feature (square root of it)
# Then, it's attached to the dataset
# The operation is done for both train and test set
X_train_new_feat = np.sqrt(X_train[:,non_linear_feat])
X_train_new_feat.shape = (X_train_new_feat.shape[0], 1)
X_train_extended = np.hstack([X_train, X_train_new_feat])

X_test_new_feat = np.sqrt(X_test[:,non_linear_feat])
X_test_new_feat.shape = (X_test_new_feat.shape[0], 1)
X_test_extended = np.hstack([X_test, X_test_new_feat])

scaler = StandardScaler()

X_train_extended_scaled = scaler.fit_transform(X_train_extended)
X_test_extended_scaled = scaler.transform(X_test_extended)
regressor = KNeighborsRegressor()
regressor.fit(X_train_extended_scaled, y_train)
y_est = regressor.predict(X_test_extended_scaled)
print "MAE=", mean_squared_error(y_test, y_est)
Out: MAE= 0.33360244905
```

Thus, we have additionally reduced the MAE and finally obtained a satisfying regressor. Of course, we may try out more methods in order to improve, but this straightforward example should influence your analysis of the application of linear and nonlinear transformations to obtain a feature that is conceptually more related to the output variable.

Dimensionality reduction

Sometimes, you will have to deal with datasets containing a large number of features, many of which may be unnecessary. This is a typical problem where you want to log as much as you can to either get enough information to properly predict the target variable, or just have more data in the future. Some features are very informative for the prediction, some are somehow related, and some are completely unrelated (that is, they only contain noise or irrelevant information).

Hence, dimensionality reduction is the operation of eliminating some features of the input dataset and creating a restricted set of features that contains all the information you need to predict the target variable in a more effective way. Reducing the number of features usually also reduces the output variability and complexity (as well as the time).

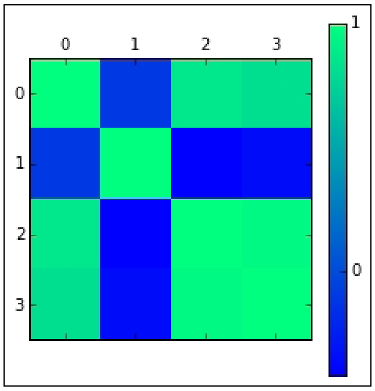
The main hypothesis behind many algorithms used in the reduction is the one pertaining to **Additive White Gaussian Noise** (**AWGN**) noise. It is an independent Gaussian-shaped noise that is added to every feature of the dataset. Reducing the dimensionality also reduces the energy of the noise since you're decreasing its span set.

The covariance matrix

The covariance matrix gives you an idea about the correlation between all the different pairs of features. It's usually the first step of dimensionality reduction because it gives you an idea of the number of features that are strongly related (and therefore, the number of features that you can discard) and the ones that are independent. On the iris dataset, where each observation has four features, this can easily be computed and understood with the help of a simple graphical representation, which can be obtained with the help of the following code:

```
In: from sklearn import datasets
import numpy as np
iris = datasets.load_iris()
cov_data = np.corrcoef(iris.data.T)
print iris.feature_names
cov_data
```

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

array([[ 1.        , -0.10936925,  0.87175416,  0.81795363],
       [-0.10936925,  1.        , -0.4205161 , -0.35654409],
       [ 0.87175416, -0.4205161 ,  1.        ,  0.9627571 ],
       [ 0.81795363, -0.35654409,  0.9627571 ,  1.        ]])

In: import matplotlib.pyplot as plt
img = plt.matshow(cov_data, cmap=plt.cm.winter)
plt.colorbar(img, ticks=[-1, 0, 1])
```



From the previous image, you can see that the value of the diagonals is 1. This is so because we're using the normalized version of the covariance matrix (normalized to the feature energy). We can also notice a high correlation of the first and third, first and fourth, and the third and fourth features. So, we see that only the second feature is almost independent of the others; the others are somehow correlated to each other.

We now have a rough idea about the potential number of features in the reduced set: 2.

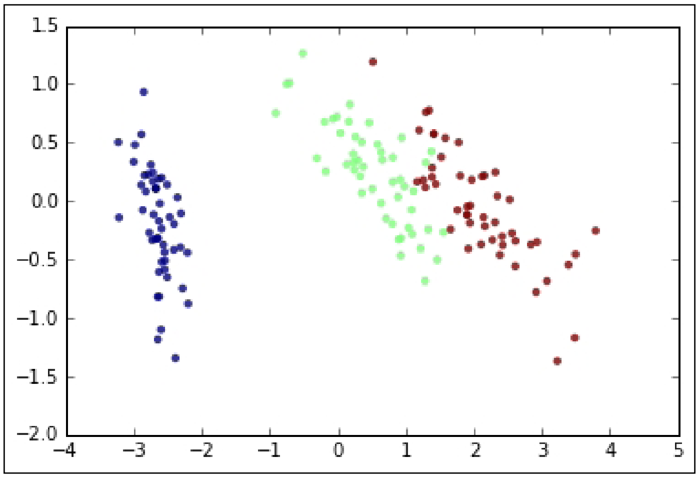
Principal Component Analysis (PCA)

PCA is a technique that helps you define a smaller and more relevant set of features. The new features are linear combinations (that is, the rotation) of the current features. After the rotation of the input space, the first vector of the output set contains most of the signal's energy (or, in other words, its variance). The second is orthogonal to the first, and it contains most of the remaining energy; the third is orthogonal to the first two vectors and contains most of the remaining energy, and so on.

In the (ideal) case of AWGN, the initial vectors contain all the information of the input signal; the ones towards the end only contain noise. Moreover, since the output basis is orthogonal, you can decompose and synthesize an approximate version of the input dataset. The key parameter, which is used to decide how many basis vectors one can use, is the energy. Since the algorithm under the hood is for singular value decomposition, eigenvectors (the basis vectors) and eigenvalues (the standard deviation associated to that vector) are the terms that are often referred to when reading about PCA. Typically, the cardinality of the output set is the one that guarantees the presence of 95 percent (or sometimes, 99 percent) of the input energy (or variance). A rigorous explanation of PCA is beyond the scope of this book, and hence, we will just inform you about the guidelines on how to use this powerful tool in Python.

Here's an example on how to reduce the dataset to two dimensions. In the previous section, we thought that two was a good number; let's see if we were right:

```
In: from sklearn.decomposition import PCA
pca_2c = PCA(n_components=2)
X_pca_2c = pca_2c.fit_transform(iris.data)
X_pca_2c.shape
Out: (150, 2)
In: plt.scatter(X_pca_2c[:,0], X_pca_2c[:,1], c=iris.target,
alpha=0.8, edgecolors='none')
plt.show()
pca_2c.explained_variance_ratio_.sum()
0.97763177502480336
```



We can immediately see that after applying the PCA, the output set has only two features. This is so because the `PCA()` object was called with the `n_components` parameter set to `2`. An alternative way to do this would be to run `PCA()` for `1`, `2`, and `3` components and then conclude that for `n_components = 2`, we got the optimum result. Then, we will see that with two basis vectors, the output dataset contains almost 98 percent of the energy of the input signal, and in the schema, the classes are pretty much neatly separable. Each color is located in a different area of the 2-dimensional Euclidean space.

> Please note that this process is automatic and you don't need to provide labels while training PCA. In fact, PCA is an unsupervised algorithm, and it does not require data related to the independent variable to rotate the projection basis.

For curious readers, the transformation matrix can be seen with the help of the following code:

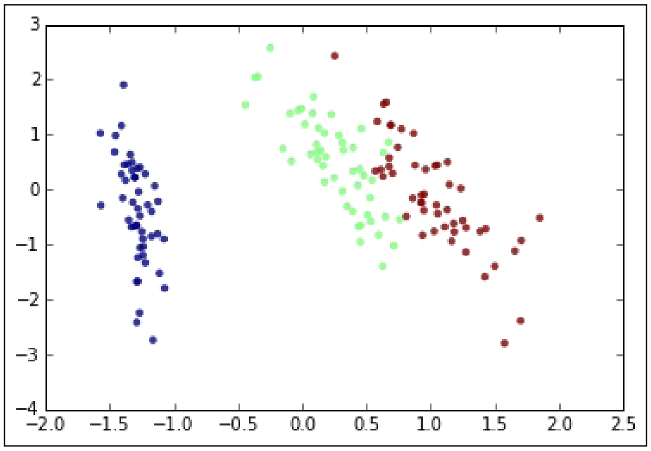
```
In: pca_2c.components_
Out:
array([[ 0.36158968, -0.08226889,  0.85657211,  0.35884393],
       [-0.65653988, -0.72971237,  0.1757674 ,  0.07470647]])
```

The transformation matrix comprises of four columns (which is the number of input features) and two rows (which is the number of the reduced ones).

Sometimes, you will find yourself in a situation where PCA is not effective enough. A possible solution for this is that you can try to whiten the signal. In this case, eigenvectors are forced to unit component-wise variances. Whitening removes information, but sometimes, it improves the accuracy of the machine learning algorithms that will be described as follows. Here's how it looks with whitening (in this case, it doesn't change anything except for the scale of the dataset with the reduced output):

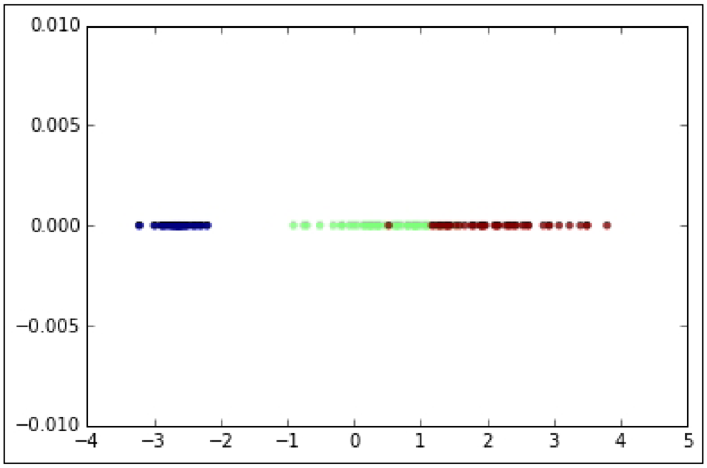
```
In: pca_2cw = PCA(n_components=2, whiten=True)
X_pca_1cw = pca_2cw.fit_transform(iris.data)
plt.scatter(X_pca_1cw[:,0], X_pca_1cw[:,1], c=iris.target, alpha=0.8, 
edgecolors='none'); plt.show()
```

```
pca_2cw.explained_variance_ratio_.sum()

Out: 0.97763177502480336
```



Now, let's try to see what happens if we project the input dataset on a 1D space generated with PCA as follows:

```
In: pca_1c = PCA(n_components=1)

X_pca_1c = pca_1c.fit_transform(iris.data)

plt.scatter(X_pca_1c[:,0], np.zeros(X_pca_1c.shape), c=iris.target,
alpha=0.8, edgecolors='none'); plt.show()

pca_1c.explained_variance_ratio_.sum()

Out: 0.92461620717426829
```



In this case, the output energy is lower (92.4 percent of the original signal), and the output points are added to the monodimensional Euclidean space. This might not be a great feature reduction step since many points with different labels are mixed together.

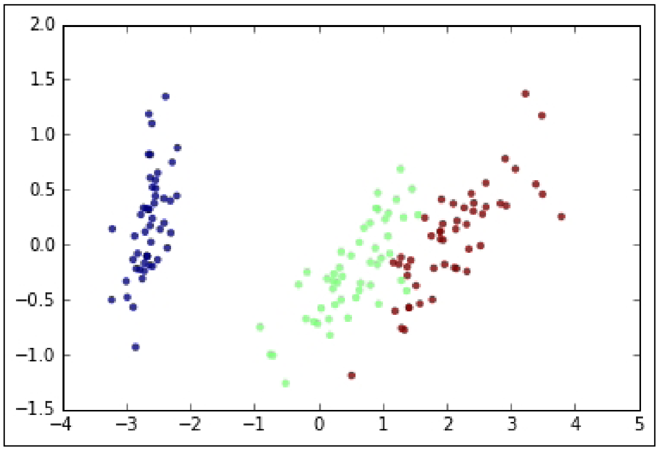
> Finally, here's a trick. To ensure that you generate an output set containing at least 95 percent of the input energy, you can just specify this value to the PCA object during its first call. A result equal to the one with two vectors can be obtained with the following code:
>
> ```
> In: pca_95pc = PCA(n_components=0.95)
> X_pca_95pc = pca_95pc.fit_transform(iris.data)
> print pca_95pc.explained_variance_ratio_.sum()
> print X_pca_95pc.shape
> Out: 0.977631775025
> (150, 2)
> ```

A variation of PCA for big data – RandomizedPCA

The main issue of PCA is the complexity of the underlying **Singular Value Decomposition** (**SVD**) algorithm. There is, anyway, a faster algorithm in Scikit-Learn based on Randomized SVD. It is a lighter but approximate iterative decomposition method. With Randomized SVD, the full-rank reconstruction is not perfect, and the basis vectors are locally optimized in every iteration. On the other hand, to get a good approximation, it requires only a few steps, making it faster than the classical SVD algorithms. Therefore, it is a great choice if the training dataset is big. In the following code, we will apply it to the iris dataset. The output is pretty close to the classical PCA since the size of the problem is very small. However, the results vary significantly when the algorithm is applied to big datasets.

```
In: from sklearn.decomposition import RandomizedPCA
rpca_2c = RandomizedPCA(n_components=2)
X_rpca_2c = rpca_2c.fit_transform(iris.data)
plt.scatter(X_rpca_2c[:,0], X_rpca_2c[:,1], c=iris.target, alpha=0.8,
edgecolors='none'); plt.show()
```

```
rpca_2c.explained_variance_ratio_.sum()
Out: 0.9776317750248038
```



Latent Factor Analysis (LFA)

Latent factor analysis is another technique that helps you reduce the dimensionality of the dataset. The overall idea is similar to PCA. However, in this case, there's no orthogonal decomposition of the input signal, and therefore, no output basis. Some data scientists think that LFA is a generalization of PCA that removes the constraint of orthogonality. Latent factor analysis is generally used when a latent factor or a construct is in the system and all the features are observations of variables of the latent factor that is linearly transformed and which has an **Arbitrary Waveform Generator** (AWG) noise. It's generally assumed that the latent factor has a Gaussian distribution and a unitary covariance. Therefore, in this case, instead of collapsing the energy/variance of the signal, the covariance among the variables is explained in the output dataset. The Scikit-learn toolkit implements an iterative algorithm, making it great for large datasets.

Here's the code to lower the dimensionality of the iris dataset by assuming two latent factors in the system:

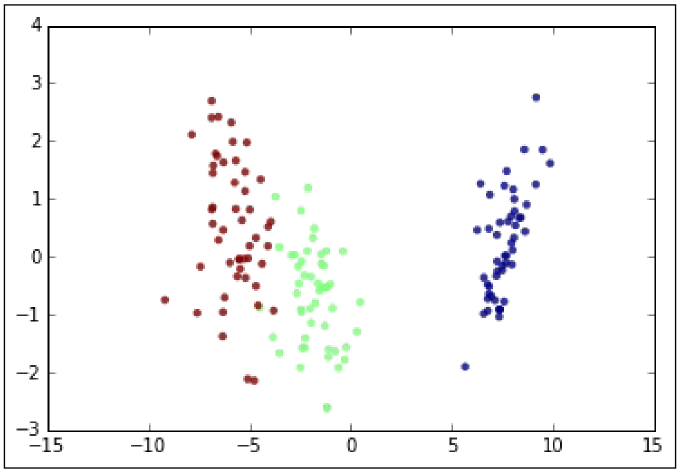
```
In: from sklearn.decomposition import FactorAnalysis
fact_2c = FactorAnalysis(n_components=2)
X_factor = fact_2c.fit_transform(iris.data)
plt.scatter(X_factor[:,0], X_factor[:,1], c=iris.target, alpha=0.8,
edgecolors='none'); plt.show()
```

Linear Discriminant Analysis (LDA)

Strictly speaking, LDA is a classifier, but it is often used for dimensionality reduction. Since it's a supervised approach, it requires the label set to optimize the reduction step. LDA outputs linear combinations of the input features, trying to model the difference between the classes that best discriminate them (since LDA uses label information). Compared to PCA, the output dataset that is obtained with the help of LDA contains neat distinction between classes. However, it cannot be used in regression problems.

Here's the application of LDA on the Iris dataset:

```
In: from sklearn.lda import LDA
lda_2c = LDA(n_components=2)
X_lda_2c = lda_2c.fit_transform(iris.data, iris.target)
plt.scatter(X_lda_2c[:,0], X_lda_2c[:,1], c=iris.target, alpha=0.8,
edgecolors='none'); plt.show()
```



Latent Semantical Analysis (LSA)

LSA is typically applied to text after it is processed with `TfidfVectorizer` or `CountVectorizer`. Compared to PCA, it applies SVD to the input dataset (which is usually a sparse matrix), producing semantic sets of words usually associated with the same concept. This is why LSA is used when the features are homogeneous (that is, all the words in the documents) and are present in large numbers.

An example of the same on Python with text and `TfidfVectorizer` is as follows. The output shows part of the content of a latent vector:

```
In: from sklearn.datasets import fetch_20newsgroups
categories = ['sci.med', 'sci.space']
twenty_sci_news = fetch_20newsgroups(categories=categories)
from sklearn.feature_extraction.text import TfidfVectorizer
tf_vect = TfidfVectorizer()
word_freq = tf_vect.fit_transform(twenty_sci_news.data)
from sklearn.decomposition import TruncatedSVD
tsvd_2c = TruncatedSVD(n_components=50)
tsvd_2c.fit(word_freq)
np.array(tf_vect.get_feature_names())[tsvd_2c.components_[20].argsort
()[-10:][::-1]]
Out: array([u'jupiter', u'sq', u'comet', u'zisfein', u'gehrels',
u'gene', u'of', u'jim', u'omen', u'theporch'],
      dtype='<U79')
```

Independent Component Analysis (ICA)

As you can guess from the name, ICA is an approach where you try to derive independent components from the input signal. In fact, ICA is a technique which allows you to create maximally independent additive subcomponents from the multivariate input signal. The main hypothesis of this technique focuses on the statistical independence of the subcomponents and their non-Gaussian distribution.

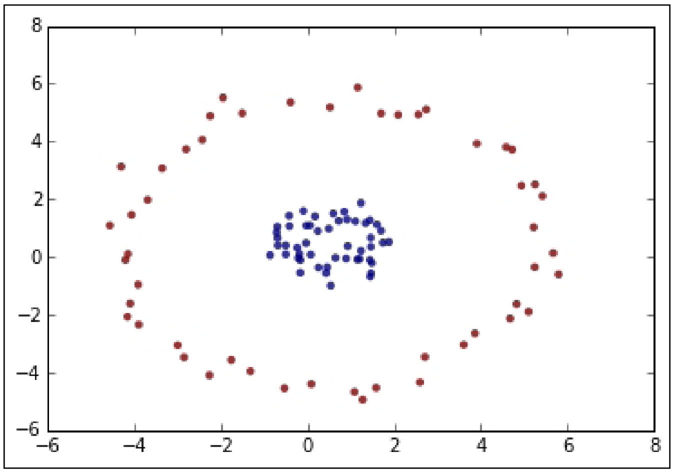
A typical scenario that may require the usage of ICA is the blind source separation. For example, two or more microphones will record two sounds (for instance, a person speaks and a song plays at the same time). In this case, ICA is able to separate the two sounds into two output features.

Scikit-learn is a faster version of the algorithm (`sklearn.decomposition.FastICA`), whose usage is similar to the other techniques.

Kernel PCA

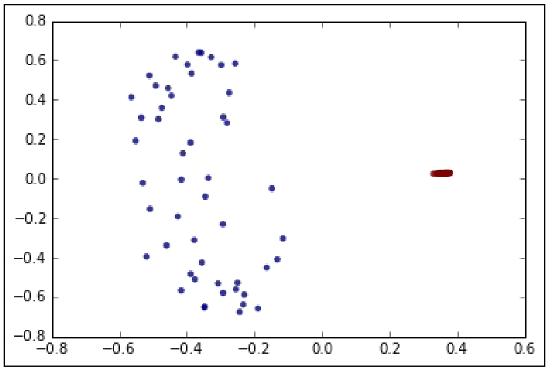
Kernel PCA is a technique that uses a kernel to map the signal on a (typically) nonlinear space and makes it linearly separable (or close to attaining the same). It's an extension of PCA, where the mapping is an actual projection on a linear subspace. There are many well-known kernels (and of course, you can always build your own on the fly), but the most used ones are: *linear*, *poly*, *RBF*, *sigmoid*, and *cosine*. They all serve different configurations of input datasets as they are able to linearize only some selected types of data. For example, let's imagine having a disk-shaped dataset, like the one created with the following code:

```
In: def circular_points (radius, N):

           return np.array([[np.cos(2*np.pi*t/N)*radius,
np.sin(2*np.pi*t/N)*radius] for t in xrange(N)])

N_points = 50

fake_circular_data = np.vstack([circular_points(1.0, N_points),
circular_points(5.0, N_points)])

fake_circular_data += np.random.rand(*fake_circular_data.shape)

fake_circular_target = np.array([0]*N_points + [1]*N_points)

plt.scatter(fake_circular_data[:,0], fake_circular_data[:,1],
c=fake_circular_target, alpha=0.8, edgecolors='none'); plt.show()
```



With this input dataset, all the linear transformations will fail to separate blue and red dots since the dataset contains circumference-shaped classes. Now, let's try this with Kernel PCA by using a RBF kernel and see what happens:

```
In: from sklearn.decomposition import KernelPCA
kpca_2c = KernelPCA(n_components=2, kernel='rbf')
X_kpca_2c = kpca_2c.fit_transform(fake_circular_data)
plt.scatter(X_kpca_2c[:,0], X_kpca_2c[:,1], c=fake_circular_target,
alpha=0.8, edgecolors='none'); plt.show()
```



Figures in this chapter can be different to the ones obtained on your local computer because graphical layout initialization is made with random parameters.

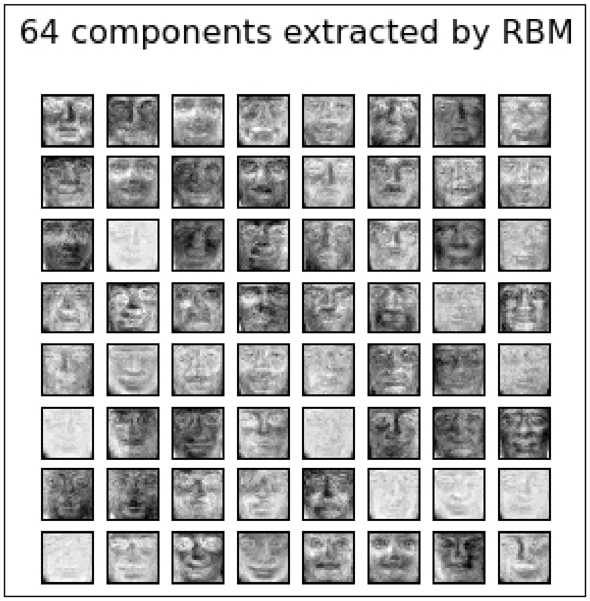
We achieved our goal—the blue dots are on the left and the red dots are on the right. Any step that follows is now easier, allowing you to deal with this dataset with linear techniques.

Restricted Boltzmann Machine (RBM)

RBM is another technique that, composed of linear functions (which are usually called hidden units or neurons), creates a nonlinear transformation of the input data. The hidden units represent the status of the system and the output dataset is actually the status of that layer.

The main hypothesis of this technique is that the input dataset is composed of features that represent probability (binary values or real values in the [0,1] range) since RBM is a probabilistic approach. In the following example, we will feed the RBM with binarized pixels of images as features (1=white, 0=black), and we will print the latent components of the system. These components represent different generic faces that appear in the original images:

```
In: from sklearn import preprocessing
from sklearn.neural_network import BernoulliRBM
n_components = 64 # Try with 64, 100, 144
olivetti_faces = datasets.fetch_olivetti_faces()
X = preprocessing.binarize(preprocessing.scale(olivetti_faces.data),
0.5)
rbm = BernoulliRBM(n_components=n_components, learning_rate=0.01,
n_iter=100)
rbm.fit(X)
plt.figure(figsize=(4.2, 4))
for i, comp in enumerate(rbm.components_):
    plt.subplot(int(np.sqrt(n_components+1)),
int(np.sqrt(n_components+1)), i + 1)
    plt.imshow(comp.reshape((64, 64)), cmap=plt.cm.gray_r,
interpolation='nearest')
    plt.xticks(()); plt.yticks(())


plt.suptitle(str(n_components) + ' components extracted by RBM',
fontsize=16)
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)
plt.show()
```



[Note that Scikit-learn contains just the base layer of RBM processing; if you are working on big datasets, you are better off using GPU-based toolkits (like the ones built on the top of CUDA or OpenCL) since RBMs are highly parallelizable.]

The detection and treatment of outliers

In data science, examples are at the core of learning from data processes. If unusual, inconsistent, or erroneous data is fed into the learning process, the resulting model may be unable to correctly generalize the accommodating of any new data. An unusually high value present in a variable may not only skew descriptive measures such as the mean and variance, but it may also distort how many algorithms learn from data, thus exposing them to unusual values and expecting unusual responses from them.

When a data point deviates markedly from the others in a sample, it is called an *outlier*. Any other expected observation is labeled as an *inlier*.

A point may be an outlier due to the following three general causes (each one implies different remedies):

- The point represents a rare occurrence, but it is yet a possible value, given the available data of the data distribution is just a sample. In such an occurrence, the generative underlying process is the same for all the points, but the outlying point may be deemed as unsuitable for a generalization due to its rarity. In such cases, the point is usually removed or underweighted. Another solution is that you can increase the sample number.
- The point represents the usual occurrence of another distribution. When similar situations occur, it is plausible to imagine an error or a misspecification that affected the generation of the sample. In any case, your learning algorithm is going to take note of a part of the distribution, which is not the focus of interest of your data science project (the focus is on the generalization). The outlier has to be removed.
- The point is clearly some kind of a mistake. For some reason, there has been a data entry error or a problem with data integrity that modified the original value and replaced it with an inconsistent value. The best course of action is to remove the value and treat it as missing at random. In this case, it is common to replace the outlier with a mean or the most common class depending on whether it is a regression or a classification problem. If it is not convenient or possible to do so, then we suggest that you just remove the example.

Univariate outlier detection

To explain the reason behind why a data point is an outlier, you are required to first locate the possible outliers in your data. There are quite a few approaches—some are univariate (you can observe each singular variable at once) and the others are multivariate (they consider more variables at the same time). The univariate methods are usually based on EDA and visualizations such as boxplots (which have been introduced at the beginning of this chapter; furthermore, it will be more specifically explained in *Chapter 6*, *Visualization*).

Basically, there are a couple of rules of thumb to keep in mind when working with single variables. Both are based on the fact that outliers may be spotted as extreme values:

If you are observing Z-scores, observations with scores higher than 3 in absolute value have to be considered as suspect outliers.

If you are observing a description of data, you can take as suspect outliers the observations that are smaller than the 25th percentile value minus the IQR (the interquartile range, that is the difference between the 75th and 25th percentile values) *1.5 and those greater that the 75th percentile value plus the IQR * 1.5. Usually, this can easily be done with the help of a boxplot graph.

In order to present how we can easily detect some outliers using Z-scores, let's load the Boston House Prices dataset. As pointed out by the description of the dataset (which you can get with the help of `boston.DESCR`), the variable CHAS, which is indexed 3, is a binary one. So, it makes little sense to use it while detecting anomalous values. In fact, such a variable can have a value of either 0 or 1:

```
In: import numpy as np
from sklearn.datasets import load_boston
boston = load_boston()
continuous_variables = [n for n in range(np.shape(boston.data)[1]) if
n!=3]
```

Now, let's quickly standardize all the continuous variables using the `StandardScaler` function from sklearn. Our target is the fancy indexing of `boston.data boston.data[:,continuous_variables]` in order to create another array containing all the variables except the previous one that was indexed 3.

`StandardScaler` automatically standardizes to zero mean and unit variance. This is a necessary routine operation that should be performed before feeding the data to the learning phase. Otherwise, many algorithms won't work properly.

Finally, let's locate the values that are above the absolute value of 3 standard deviations:

```
In:
from sklearn import preprocessing
normalized_data =
preprocessing.StandardScaler().fit_transform(boston.data[:,continuous
_variables])
outliers_rows, outliers_columns = np.where(np.abs(normalized_data)>3)
```

The outliers_rows and outliers_columns variables contain the row and column indexes of the suspect outliers. We can print the index of the examples:

```
In: print outliers_rows
Out: [ 55  56  57 102 141 199 200 201 202 203 204 225 256 257 262 283
284 347 …
```

Alternatively, we can display the tuple of the row/column coordinates in the array:

```
In: print(list(zip(outliers_rows, outliers_columns)))
Out: [(55, 1), (56, 1), (57, 1), (102, 10), (141, 11), (199, 1),
(200, 1), (201, 1), ...
```

The univariate approach can reveal quite a lot of potential outliers. Anyway, it won't disclose an outlier that does not have an extreme value. However, it will reveal the outlier if it finds an unusual combination of values in two or more variables. Often in such cases, the values of the involved variables may not even be extreme, and therefore, the outlier may slip away unnoticed.

In order to discover cases where this happens, you can use a dimensionality reduction algorithm, such as the previously illustrated PCA, and then check the absolute values of the components that are beyond three standard deviations.

Anyway, the Scikit-learn package offers a couple of classes that can automatically work for you straight out of the box and signal all suspect cases:

- The covariance.EllipticEnvelope class fits a robust distribution estimation of your data, pointing out the outliers that might be contaminating your dataset because they are the extreme points in the general distribution of the data.

- The `svm.OneClassSVM` class is a support vector machine algorithm that can approximate the shape of your data and find out if any new instances provided should be considered as a novelty (it acts as a novelty detector because by default, it presumes that there is no outlier in the data). Anyway, by just modifying its parameters, it can also work on a dataset where outliers are present, providing an even more robust and reliable outlier detection system than `EllipticEnvelope`.

Both classes, based on different statistical and machine learning approaches, need to be known and applied during your modeling phase.

EllipticEnvelope

`EllipticEnvelope` is a function that tries to figure out the key parameters of your data's general distribution by assuming that your entire data is an expression of an underlying multivariate Gaussian distribution. Simplifying as much as possible the complex estimations working behind the algorithm, we can say that it checks the distance of each observation with respect to a grand mean that takes into account all the variables in your dataset. For this reason, it is able to spot both univariate and multivariate outliers.

The only parameter that you have to take into account when using this function from the covariance module is the contamination parameter, which can take a value of up to 0.5. It provides information to the algorithm about the proportion of the outliers present in your dataset. Situations may vary from dataset to dataset. However, as a starting figure, we suggest a value from 0.01–0.02 since it is the percentage of observations that should fall over the absolute value 3 in the Z score distance from the mean in a standardized normal distribution. For this reason, we deem the default value of 0.1 as too high.

Let's see this algorithm in action with the help of a synthetic distribution:

```
In: # Create an artificial distribution made of blobs
from sklearn.datasets import make_blobs
blobs = 1
blob = make_blobs(n_samples=100, n_features=2, centers=blobs,
cluster_std=1.5, shuffle=True, random_state=5)
# Robust Covariance Estimate
```

```
from sklearn.covariance import EllipticEnvelope
robust_covariance_est = EllipticEnvelope(contamination=.1).fit(blob[0])
detection = robust_covariance_est.predict(blob[0])
outliers = np.where(detection==-1)
inliers = np.where(detection==1)
# Draw the distribution and the detected outliers
import matplotlib.pyplot as plt# Just the distribution
plt.plot(blob[0][:,0],blob[0][:,1], 'x', markersize=10,
color='black', alpha=0.8)
plt.show()
# The distribution and the outliers
a = plt.plot(blob[0][inliers,0],blob[0][inliers,1], 'x',
markersize=10, color='black', alpha=0.8, label='inliers')
b = plt.plot(blob[0][outliers,0],blob[0][outliers,1], 'o',
markersize=6,color='black', alpha=0.8, label='outliers')
plt.legend((a[0],b[0]),('inliers','outliers'),numpoints=1,loc='lower
right')
plt.show()
```

Let's examine this code closely.

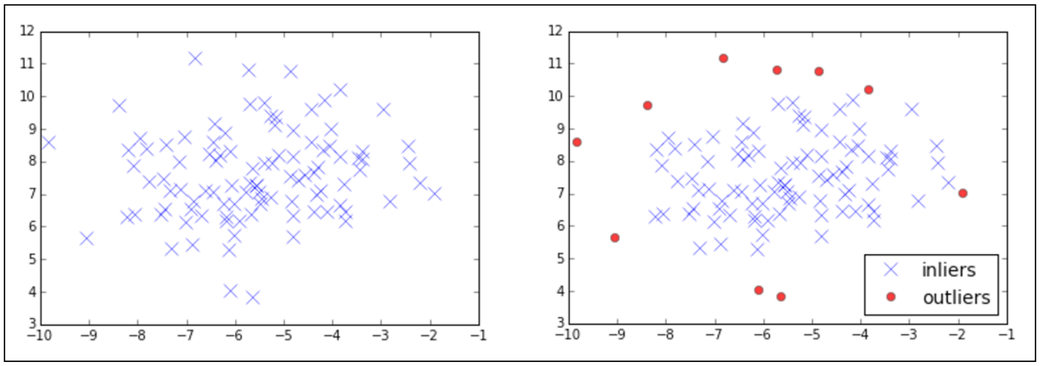
The `make_blobs` function creates a certain number of distributions into a bidimensional space for a total of 100 examples (the `n_samples` parameter). The number of distributions (parameter centers) is related to the user-defined variable blobs, which is initially set to 1.

After creating the artificial example data, running `EllipticEnvelope` with a contamination rate of 10 percent helps you find out the most extreme values in the distribution. The model deploys first fit by using the `.fit()` method on the `EllipticEnvelope` class. Then, a prediction is obtained by using the `.predict()` method on the data that was used for the fit.

The results, corresponding to a vector of values 1 and -1 (with -1 being the mark for anomalous examples) can be displayed thanks to a couple of scatterplots using the `plot` function from the `pyplot` module in `matplotlib`.

The distinction between inliers and outliers is recorded in the variable's outliers and inliers, which contain the indexes of the examples.

Now, let's run the code a few more times after changing the number of blobs and examine the results when the blobs have a value of `1` and `4`:



In the case of a unique underlying multivariate distribution (when the variable blobs = 1), the EllipticEnvelope algorithm has successfully located 10 percent of the observations on the fringe of the distribution itself and has consequently signaled all the suspect outliers.

Instead, when multiple distributions are present in the data as if there were two or more natural clusters, the algorithm, trying to fit a unique general distribution, tends to locate the potential outliers on just the most remote cluster, thus ignoring other areas of data that might be potentially affected by outlying cases.

This is not an unusual situation with real data, and it represents an important limitation of the EllipticEnvelope algorithm.

Now, let's get back to our initial Boston house price dataset for verification of some more data that is more realistic than our synthetic blobs. Here is the first part of the code that we can use for our experiment:

```
In: from sklearn.decomposition import PCA
# Normalized data relative to continuous variables
continuous_variables = [n for n in range(np.shape(boston.data)[1]) if
n!=3]
normalized_data =
preprocessing.StandardScaler().fit_transform(boston.data[:,continuous
_variables])
# Just for visualization purposes pick the first 2 PCA components
pca = PCA(n_components=2)
Zscore_components = pca.fit_transform(normalized_data)
```

```
vtot = 'PCA Variance explained ' +
str(round(np.sum(pca.explained_variance_ratio_),3))
v1 = str(round(pca.explained_variance_ratio_[0],3))
v2 = str(round(pca.explained_variance_ratio_[1],3))
```

In this script, we will first standardize the data and then, just for subsequent visualization purposes, generate a reduction to two components by using PCA.

The two PCA components account for about 62 percent of the initial variance expressed by the 12 continuous variables available in the dataset (the summed value of the `.explained_variance_ratio_` variable that is internal to the fitted `PCA` class).

Although only two PCA components are sufficient for visualization purposes of this example, normally you'd get more than two components for this dataset since the target is to have enough to account for at least 95 percent of the total variance (as stated previously in the chapter).

We will continue with the script:

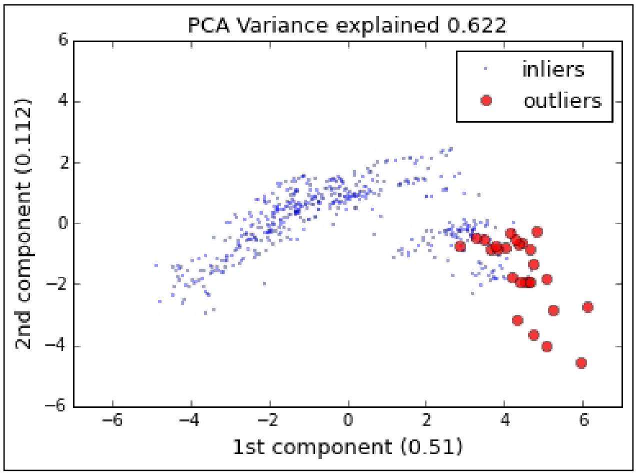
```
In: # Robust Covariance Estimate
robust_covariance_est = EllipticEnvelope(store_precision=False,
assume_centered = False, contamination=.05)
robust_covariance_est.fit(normalized_data)
detection = robust_covariance_est.predict(normalized_data)
outliers = np.where(detection==-1)
regular = np.where(detection==1)

# Draw the distribution and the detected outliers
from matplotlib import pyplot as plt
a = plt.plot(Zscore_components[regular,0],Zscore_components[regular,1],
'x', markersize=2, color='black', alpha=0.6, label='inliers')
b = plt.plot(Zscore_components[outliers,0],Zscore_components[outliers,1],
'o', markersize=6,color='black', alpha=0.8, label='outliers')
plt.xlabel('1st component ('+v1+')')
plt.ylabel('2nd component ('+v2+')')
plt.xlim([-7,7])
plt.ylim([-6,6])
plt.legend((a[0],b[0]),('inliers','outliers'),numpoints=1,loc='best')
plt.title(vtot)
plt.show()
```

As in the previous example, the code estimates that `EllipticEnvelope` (assuming a low contamination that is equivalent to 0.05) predicts the outliers and stores them in an array in the same way as it stores the inliers. Finally, there's the visualization (as mentioned before, we are going to discuss all the visualization methods in *Chapter 6*, *Visualization*).

Now, let's observe the result offered by the scatterplot we generated to visualize the first two `PCA` components of the data and mark the outlying observations.

From a general point of view in regard to the general distribution of the point, as provided by the two components that account for about 62 percent of the variance in the data, it really seems as if there should be two main distinct clusters of house prices in Boston that correspond to the high- and low-end units in the market. This is surely a nonoptimal situation for `EllipticEnvelope` estimations.

Anyway, in line with what we noticed while experimenting with the synthetic blobs, the algorithm in this instance has pointed out the outliers on just a cluster—the lesser one. Given such results, there is a strong reason to believe that we just received a partial response and some further investigation will be required for the same. The Scikit-learn package actually integrates the robust covariance estimation method, which is basically a statistical approach, with another methodology that is well-rooted in machine learning, the OneClassSVM class. Now, we shall move on to experiment with it.



> Before closing this experiment, please also note that to fit both PCA and EllipticEnvelope, we used an array named `normalized_data`, which contains just the standardized continuous dataset variables. Please always consider in your projects that using nonstandardized data and mixing binary or categorical data with continuous ones may induce errors and approximate estimations for the EllipticEnvelope algorithm.

OneClassSVM

As EllipticEnvelope fits a hypothetical Gaussian distribution, leveraging parametric, statistical assumption, OneClassSVM is a machine learning algorithm that learns from the data what the data distribution should be, and it is therefore applicable in a bigger variety of datasets.

It is great if you have a clean dataset and have it fit perfectly. Afterwards, OneClassSVM can be summoned to check if any new example fits in the historical distribution, and if it doesn't, it will signal a novel example, which might be both an error or some new, previously unseen situation.

Just think of data science situations as a machine learning classification algorithm trained to recognize posts and news on a Website and take online actions. OneClassSVM can easily spot a post that is different from the others present on the Website (spam, maybe?), whereas other algorithms will just try to fit the new example into the existing topic categorization.

However, OneClassSVM can also be used to spot existing outliers. If a distribution could not be modeled by this SVM class and it lies at the margins, then there is surely something fishy about it.

In order to have it work as an outlier detector, you need to work on its core parameters. OneClassSVM requires you to define the kernel, degree, gamma, and nu.

- **Kernel and degree**: They are interconnected. Usually, the values that we suggest on the basis of our experience are the default ones; the value of kernel should be `rbf` and its degree should be 3. Such parameters will inform OneClassSVM to create a series of classification bubbles that span through three dimensions, allowing you to model even the most complex multidimensional distribution forms.

- **Gamma**: It is a parameter connected to the rbf kernel. We suggest that you keep it as low as possible. A good rule of thumb should be to assign it a minimum value that lies between the inverse of the number of cases and the variables. Gamma will be extensively explained in *Chapter 4*, *Machine Learning*. Anyway, it will suffice for now to say that higher values of gamma tend to lead the algorithm to follow the data more to define the shape of the classification bubbles.
- **Nu**: It is the parameter that determines whether we have to fit the exact distribution or we should try to keep a certain generalization by not adapting too much to the present data (a necessary choice if outliers are present). It can be easily determined with the help of the following formula:

  *nu_estimate = 0.95 * outliers_fraction + 0.05*
- If the value of the outliers fraction is very small, nu will be small and the SVM algorithm will try to fit the contour of the data points. On the other hand, if the fraction is high, so will be the parameter, forcing a smoother boundary of the inliers' distributions.

Let's immediately observe the performance of this algorithm on the problem that we faced before on the Boston house price dataset:

```
In: from sklearn.decomposition import PCA
from sklearn import preprocessing
from sklearn import svm
# Normalized data relative to continuous variables
continuous_variables = [n for n in range(np.shape(boston.data)[1]) if
n!=3]
normalized_data =
preprocessing.StandardScaler().fit_transform(boston.data[:,continuous
_variables])
# Just for visualization purposes pick the first 5 PCA components
pca = PCA(n_components=5)
Zscore_components = pca.fit_transform(normalized_data)
vtot = 'PCA Variance explained ' +
str(round(np.sum(pca.explained_variance_ratio_),3))
# OneClassSVM fitting and estimates
outliers_fraction = 0.02 #
```

```
nu_estimate = 0.95 * outliers_fraction + 0.05
machine_learning = svm.OneClassSVM(kernel="rbf",
gamma=1.0/len(normalized_data), degree=3, nu=nu_estimate)
machine_learning.fit(normalized_data)
detection = machine_learning.predict(normalized_data)
outliers = np.where(detection==-1)
regular = np.where(detection==1)
# Draw the distribution and the detected outliers
from matplotlib import pyplot as plt
for r in range(1,5):
  a =
plt.plot(Zscore_components[regular,0],Zscore_components[regular,r],
'x', markersize=2, color='blue', alpha=0.6, label='inliers')
  b =
plt.plot(Zscore_components[outliers,0],Zscore_components[outliers,r],
'o', markersize=6,color='red', alpha=0.8, label='outliers')
  plt.xlabel('Component 1
('+str(round(pca.explained_variance_ratio_[0],3))+')')
  plt.ylabel('Component
'+str(r+1)+'('+str(round(pca.explained_variance_ratio_[r],3))+')')
  plt.xlim([-7,7])
  plt.ylim([-6,6])

plt.legend((a[0],b[0]),('inliers','outliers'),numpoints=1,loc='best')
  plt.title(vtot)
  plt.show()
```

Compared to the code presented before, this code differentiates because a PCA is made up of five components in order to explore more data dimensions. Another reason for this is the usage of OneClassSVM.

The core parameters are calculated from the number of observations, as follows:

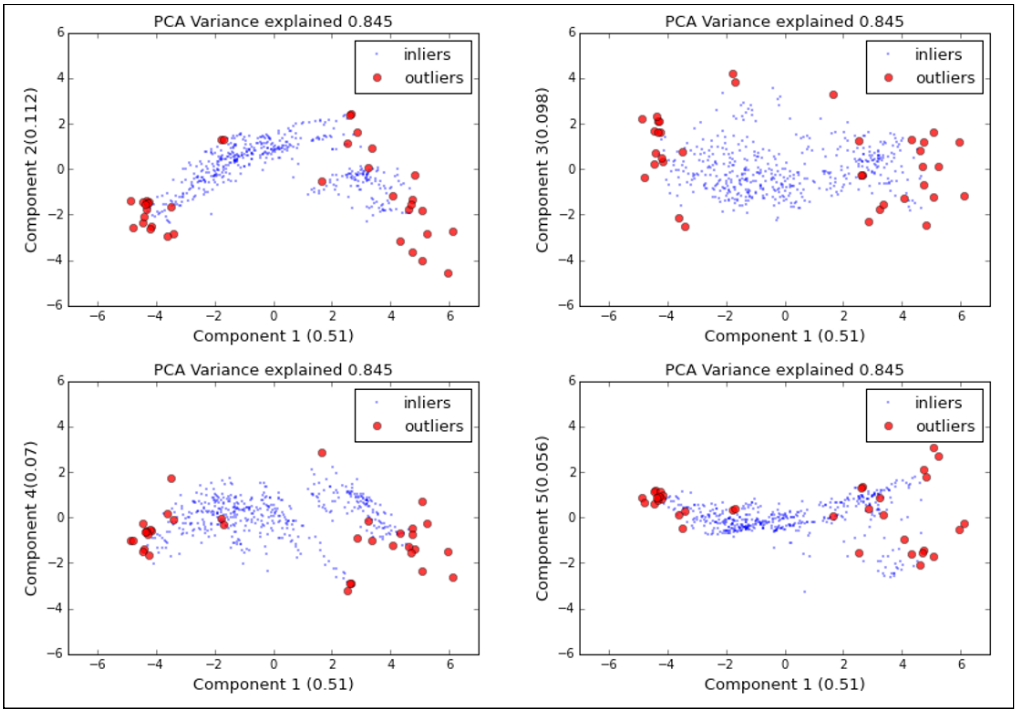
- `gamma=1.0/len(normalized_data)`
- `nu=nu_estimate`

In particular, nu depends on:

*nu_estimate = 0.95 * outliers_fraction + 0.05*

So, by changing `outliers_fraction` (from 0.02 to a larger value, such as 0.1), you may experience the result when supposing a larger incidence of anomalous cases in your data.

Let's also observe the graphical output of different components from two to five and compare it with the principal component (51 percent of explained variance). The first one is as follows:



It looks like this approach modeled the distribution of house price data better and spotted a few extreme values on the borders of the distribution.

At this point, you can decide on one of the two novelties and outlier detection methods. You may even use both in order to do the following:

- Further scrutinize the characteristics of the outliers in order to figure out a reason for them (which could furthermore make you reflect on the generative processes underlying your data)
- Trying to build machine learning models; including under-weighting, or excluding the outlying observations

In the end, in a pure data science approach, what will help you decide what to do with outlying observations is the testing of the results of your decisions and consequent operations on data. This is a topic that we are going to discuss with you in the upcoming sections.

Scoring functions

In order to evaluate the performance of the system and check how close you are to the objective that you have in mind, you need to use a function that scores the outcome. Typically, different scoring functions are used to deal with binary classification, multilabel classification, regression, or a clustering problem. Now, let's see the most popular functions for each of these tasks.

Multilabel classification

When your task is to predict more than a single label (for instance, what's the weather like today? Which flower is this? What's your job?), it's called a multilabel classification. This is a very popular task, and many performance metrics exist to evaluate classifiers. Of course, you can use all these measures in the case of binary classification. Now, let's explain them with a simple real-world example:

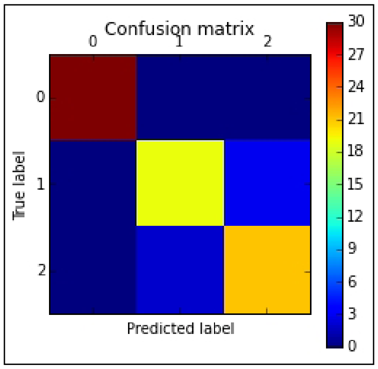
```
In: from sklearn import datasets
iris = datasets.load_iris()
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(iris.data,
iris.target, test_size=0.50, random_state=4)
# Use a very bad multiclass classifier
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(max_depth=2)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
iris.target_names
Out: ['setosa', 'versicolor', 'virginica'],
dtype='|S10')
```

Now, let's take a look at the measures used in multilabel classification:

- **Confusion matrix**: Before we describe the performance metrics for multilabel classification, let's take a look at **Confusion matrix**, a table that gives us an idea about what the misclassifications are for each class. Ideally, in a perfect classification, all the cells that are not on the diagonal should be 0s.

  In the following example, you will instead see that class 0 (setosa) is never misclassified, class 1 (versicolor) is misclassified thrice as virginica, and class 2 (virginica) is misclassified twice as versicolor:

  ```
  In: from sklearn import metrics
  from sklearn.metrics import confusion_matrix
  cm = confusion_matrix(y_test, y_pred)
  print cm
  Out: [[30  0  0]
        [ 0 19  3]
        [ 0  2 21]]
  ```



- **Accuracy**: Accuracy is the portion of the predicted labels that are exactly equal to the real ones. In other words, it's the percentage of correctly classified labels.

  ```
  In: print "Accuracy:", metrics.accuracy_score(y_test, y_pred)
  Accuracy: 0.933333333333
  ```

- **Precision**: It is a measure that is taken from the information retrieval world. It counts the number of relevant results in the result set. Equivalently, in a classification task, it counts the number of correct labels in each set of classified labels. Results are then averaged on all the labels:

  ```
  In: print "Precision:", metrics.precision_score(y_test, y_pred)
  Precision: 0.933333333333
  ```

- **Recall**: It is another concept taken from information retrieval. It counts the number of relevant results in the result set, compared to all the relevant labels in the dataset. In classification tasks, that's the amount of correctly classified labels in the set divided by the total count of labels for that set. Results are then averaged in the following way:

  ```
  In: print "Recall:", metrics.recall_score(y_test, y_pred)
  Recall: 0.933333333333
  ```

- **F1 Score**: It is a harmonic average of precision and recall.

  ```
  In: print "F1 score:", metrics.f1_score(y_test, y_pred)
  F1 score: 0.933267359393
  ```

These are the most used measures in multilabel classification. There is a convenient method that shows a report on these measures, which is very handy. Support is simply the number of observations with that label. It's pretty useful to understand whether a dataset is balanced (that is, whether it has the same support for every class) or not.

```
In: from sklearn.metrics import classification_report
print classification_report(y_test, y_pred,
target_names=iris.target_names)
Out:
             precision    recall  f1-score   support
     setosa       1.00      1.00      1.00        30
 versicolor       0.90      0.86      0.88        22
  virginica       0.88      0.91      0.89        23
avg / total       0.93      0.93      0.93        75
```

In practice, Precision and Recall are used more extensively than Accuracy as most datasets in DS tend to be unbalanced. To account for this imbalance, data scientists often present their results in terms of Precision, Recall, and F1 tuple.

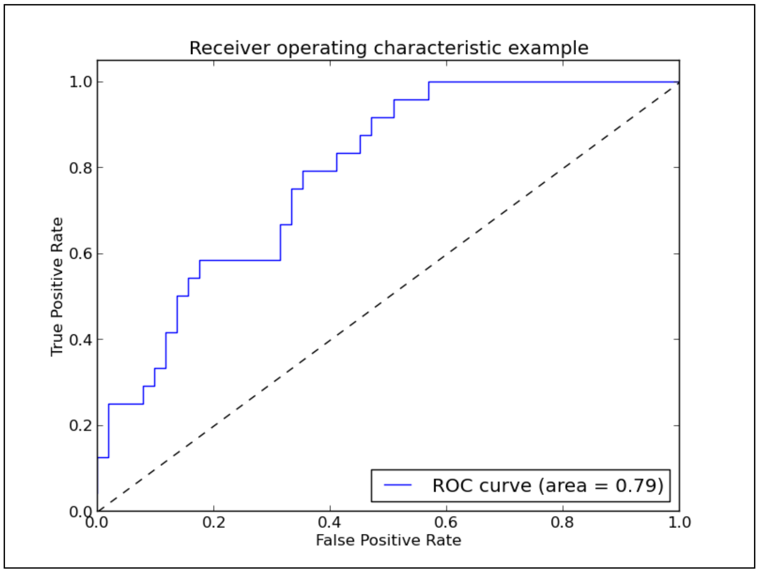
Also, Accuracy, Precision, Recall, and F1 assume values in the [0.0, 1.0] range. Perfect classifiers achieve the score of 1.0 for all these measures.

Binary classification

In addition to the ones shown in the preceding section, in problems where you have only two output classes (for instance, if you have to guess the gender or predict whether the user will click/buy/like the item), there are additional measures. The most used, since it's very informative, is the area under **Receiver Operating Characteristics curve** (**ROC**) or **area under a curve** (**AUC**).

The ROC curve is a graphical way to express how the performances of the classifier change over all the possible classification thresholds (that is, the change in outcomes when its parameters change.) Specifically, the performances have a true positive (or hit) rate, and a false positive (or miss) rate. The first is the rate of the correct positive results, and the second is the rate of the incorrect ones. The area under that curve represents how well the classifier performs with respect to a random classifier (whose AUC is 0.50).

Here, we have a graphical example of a random classifier (dotted line) and a better one (solid line). You can see that the AUC of a random classifier is 0.5 (it is half of the square), and the other has a higher AUC (with its upper bounded to `1.0`):



The function that is used to compute the AUC with Python is `sklearn.metrics.roc_auc_score()`.

Regression

In tasks where you have to predict real numbers or regression, many measures' functions are derived from Euclidean algebra.

- **Mean absolute error or MAE**: It is the mean L1 norm of the difference vector of the predicted and real values.

```
In: from sklearn.metrics import mean_absolute_error
mean_absolute_error([1.0, 0.0, 0.0], [0.0, 0.0, -1.0])
```

```
Out: 0.66666666666666663
```

- **Mean squared error or MSE**: It is the mean L2 norm of the difference vector of the predicted and real values.

```
In: from sklearn.metrics import mean_squared_error
mean_squared_error([-10.0, 0.0, 0.0], [0.0, 0.0, 0.0])
```

```
Out: 33.3333333333333
```

- **R2 score**: R2 is also known as the Coefficient of Determination. In a nutshell, R2 determines how good a linear fit is that exists between the predictors and the target variable. It takes values between 0 and 1 (inclusive); the higher it is, the better the model. There are many more intricacies about this metric that you can find in a reference book on Statistics. The function to use in this case is `sklearn.metrics.r2_score`.

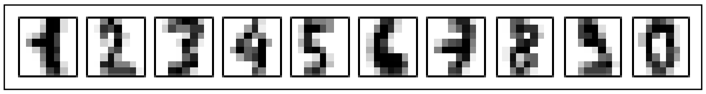
Testing and validating

After loading our data, preprocessing it, creating new useful features, checking for outliers and other inconsistent data points, and choosing the right metric, we are finally ready to apply some machine learning algorithm that, by observing a series of examples and pairing them with their outcome, is able to extract a series of rules that can be successfully generalized to new examples by correctly guessing their resulting outcome. This is the supervised learning approach where a series of specialized algorithms that are fundamental to data science is used. How can we correctly apply the learning process in order to achieve the best generalizable model for prediction?

There are some best practices to be followed. Let's proceed step by step, by first loading the dataset that we will be working on in the following example:

```
In: from sklearn.datasets import load_digits
digits = load_digits()
print digits.DESCR
X = digits.data
y = digits.target
```

The digit dataset contains images of handwritten numbers from zero to nine. The data format consists of a matrix of 8x8 images of this kind:



These digits are actually stored as a vector (resulting from the flattening of each 8x8 image) of 64 numeric values from 0 to 16, representing greyscale tonality for each pixel:

```
In: X[0]
Out: array([0., 0., 5., 13., 9., 1., 0., 0., …])
```

We will also upload three different machine learning hypotheses and three support vector machines for classification. They will also be useful for our practical example:

```
In: from sklearn import svm
h1 = svm.LinearSVC(C=1.0) # linear SVC
h2 = svm.SVC(kernel='rbf', degree=3, gamma=0.001, C=1.0) # Radial
basis SVC
h3 = svm.SVC(kernel='poly', degree=3, C=1.0) # 3rd degree polynomial
SVC
```

As a first experiment, let's fit the linear SVC to our data and verify the results:

```
In: h1.fit(X,y)
print h1.score(X,y)
Out: 0.9938786867
```

The first method fits a model using the `X` array in order to correctly predict one of the 10 classes indicated by the `y` vector. After that, by calling the `.score()` method and specifying the same predictors (the `X` array), the method evaluates the performance in terms of mean accuracy with respect to the true values given by the `y` vector. The result is about 99.4 percent accurate in predicting the correct digit.

This number represents the in-sample performance, which is the performance of the learning algorithm. It is purely indicative, though it represents an upper bound of the performance (providing different examples, the average performance will always be inferior). In fact, every learning algorithm has a certain capability to memorize the data with which it has been trained. So, the in-sample performance is partly due to the capability of the algorithm to learn some general inference from the data, and partly from its memorization capabilities. In extreme cases, if the model is either overtrained or too complex with respect to the available data, the memorized patterns prevail over the derived rules and the algorithm becomes unfit to correctly predict new observations. This problem is called overfitting. Since we cannot separate these two concomitant effects, in order to have a proper estimate of the predictive performances of our hypothesis, we do need to test it on some fresh data where there is no memorization effect.

Memorization happens because of the complexity of the algorithm; complex algorithms own many coefficients, where information about the data can be stored. Unfortunately, the memorization effect causes high variance in the estimation of unseen cases since its predictive processes become random. Three solutions are possible.

Firstly, you can increase the number of examples so that it will become infeasible to store information about the previously seen cases, but it may become more expensive to find all the necessary amount of data.

Secondly, you can use a simpler machine learning algorithm and thus become not only less prone to memorization, but also less capable of fitting the complexity of the rules underlying the data.

Thirdly, you can use regularization to penalize extremely complex models and force the algorithm to underweight, or even exclude, a certain number of variables.

In many cases, fresh data is not available, if not at a certain cost. A good approach would be to divide the initial data into a training set (usually 70-80 percent of the total data) and a test set (the remaining 20-30 percent). The split between the training and the test set should be completely random, taking into account any possible unbalanced class distribution:

```
In: from sklearn import cross_validation
chosen_random_state = 1
X_train, X_test, y_train, y_test =
cross_validation.train_test_split(X, y, test_size=0.30,
random_state=chosen_random_state)
print "X train shape %s, X test shape %s, \ny train shape %s, y test
shape %s" % (X_train.shape, X_test.shape, y_train.shape,
y_test.shape)
h1.fit(X_train,y_train)
print h1.score(X_test,y_test) # Returns the mean accuracy on the
given test data and labels
Out: X train shape (1257L, 64L), X test shape (540L, 64L),
y train shape (1257L,), y test shape (540L,)
0.953703703704
```

By executing the following code, the initial data is randomly split into two sets thanks to the `cross_validation.train_test_split()` function on the basis of `test_size` (which could be an integer indicating the exact number of examples for the test set or a float, indicating the percentage of the total data to be used for testing purposes). The split is governed by `random_state`, which assures that the operation is reproducible at different times and on different computers (even when you're using completely different operating systems).

The present average accuracy is 0.94. Let's try to run the same cell again that changes, with every run, the integer value of the `chosen_random_state` and note how the average accuracy will actually change, signaling us that the test set is also not an absolute measure of performance and that it should be used.

Actually, we can get a biased performance estimation from the test set if we either choose (after various trials with `random_state`) a test set that can confirm our hypothesis, or start using the test set as a reference to take decisions in regard to the learning process (for example, selecting the best hypothesis that fits a certain test sample).

The resulting performance would surely look better, but it would not be a representative of the real performance of the machine learning system that we have built.

Therefore, when we have to choose between multiple hypotheses (a common experiment in data science) after fitting each of them onto the training data, we need a data sample that can be used to compare their performances, and it cannot be the test set (because of the reasons that we mentioned previously).

A correct approach is to use a validation set. We suggest that you split the initial data; 60 percent of the initial data can be reserved for the training set, 20 percent for the validation set, and 20 percent for the test set. Our initial code can be changed to take this into account and it can be adapted to test all the three hypotheses:

```
In: chosen_random_state = 1

X_train, X_validation_test, y_train, y_validation_test =
cross_validation.train_test_split(X, y, test_size=.40,


random_state=chosen_random_state)

X_validation, X_test, y_validation, y_test =
cross_validation.train_test_split(X_validation_test,
y_validation_test,


test_size=.50, random_state=chosen_random_state)

print "X train shape, %s, X validation shape %s, X test shape %s, \ny
train shape %s, y validation shape %s, y test shape %s\n" %
(X_train.shape, X_validation.shape, X_test.shape, y_train.shape,
y_validation.shape, y_test.shape)

for hypothesis in [h1, h2, h3]:

    hypothesis.fit(X_train,y_train)

    print "%s -> validation mean accuracy = %0.3f" % (hypothesis,
hypothesis.score(X_validation,y_validation))

h2.fit(X_train,y_train)
```

```
print "\n%s -> test mean accuracy = %0.3f" % (h2,
h2.score(X_test,y_test))


Out:
X train shape, (1078L, 64L), X validation shape (359L, 64L), X test
shape (360L, 64L),
y train shape (1078L,), y validation shape (359L,), y test shape
(360L,)


LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='l2', multi_class='ovr', penalty='l2',
     random_state=None, tol=0.0001, verbose=0) -> validation mean
accuracy = 0.964
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False) ->
validation mean accuracy = 0.992
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.0,
  kernel='poly', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False) -> validation mean
accuracy = 0.989


SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False) ->
test mean accuracy = 0.978
```

As reported by the output, the training set is now made up of 1078 cases (60 percent of the total cases). In order to divide the data in three parts—training, validation, and test—at first, the data is divided using the `cross_validation.train_test_split` function between the train set and a test/validation dataset. Then, the test/validation dataset is split into two parts using the same function. Each hypothesis, after being trained, is tested against the validation set. The SVC with an RBF kernel, by obtaining a mean accuracy of 0.992, is the best model according to the validation set. Having decided to use this model, its performance was further evaluated on the test set, resulting in an accuracy of 0.978.

Since the test's accuracy is different from that of the validation one, is the chosen hypothesis really the best one? We suggest that you try to run the code in the cell multiple times (ideally, running the code about 30 times should ensure statistical significance), each time changing the `chosen_random_state` value. The same learning procedure will therefore be validated with respect to different samples.

Cross-validation

If you have run the previous experiment, you may have realized that:

1. Both the validation and test results vary as samples are different
2. The chosen hypothesis is often the best one, but this is not always the case

Unfortunately, relying on the validation and testing phases of samples brings uncertainty along with a strong reduction of the learning examples for training (the fewer the examples, the more the variance of the obtained model).

A solution is to use cross-validation, and Scikit-learn offers a complete module for cross-validation and performance evaluation (`sklearn.cross_validation`).

By resorting to cross-validation, you'll just need to separate your data into a training and test set, and you will be able to use the training data for both model optimization and model training.

How does cross-validation work? The idea is to divide your training data into a certain number of partitions (called folds) and train your model as many times as the number of partitions, keeping out of training a different partition every time. Ten folds is quite a common configuration that even we recommend. After every model training, you will test the result on the fold that is left out and store it away. In the end, you will have as many results as folds, and you can calculate both the average and standard deviation on them. The standard deviation will provide a hint on how your model is influenced by the data that is provided for training (the variance of the model, actually), and the mean will provide a fair estimate of its general performance. Using the mean of the cross-validation results of different models (different because of the model type, used selection of the training variables, or the model's hyper-parameters), you can confidently choose the best performing hypothesis to be tested for general performance.

> We strongly suggest that you use cross-validation just for optimization purposes and not for performance estimation (that is, to figure out what the error of the model might be on fresh data). Cross-validation just points out the best possible algorithm and parameter choice on the basis of the best averaged result. Using it for performance estimation would mean using the best result found. In order to report an unbiased estimation of your possible performance, you should prefer using a test set.

Let's execute an example in order to see cross-validation in action. At this point, we can review out the previous evaluation of three possible hypotheses for our digits dataset:

```
In: choosen_random_state = 1

cv_folds = 10 # Try 3, 5 or 20

eval_scoring='accuracy' # Try also f1

workers = -1 # this will use all your CPU power

X_train, X_test, y_train, y_test =
cross_validation.train_test_split(X, y, test_size=0.30,
random_state=choosen_random_state)

for hypothesis in [h1, h2, h3]:

    scores = cross_validation.cross_val_score(hypothesis, X_train,
y_train, cv=cv_folds, scoring= eval_scoring, n_jobs=workers)

    print "%s -> cross validation accuracy: mean = %0.3f std = %0.3f"
% (hypothesis, np.mean(scores), np.std(scores))
```

```
Out:

LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,

     intercept_scaling=1, loss='l2', multi_class='ovr', penalty='l2',

     random_state=None, tol=0.0001, verbose=0) -> cross validation
accuracy: mean = 0.938 std = 0.015

SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,

  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,

  random_state=None, shrinking=True, tol=0.001, verbose=False) ->
cross validation accuracy: mean = 0.990 std = 0.007

SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.0,
  kernel='poly', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False) -> cross validation
accuracy: mean = 0.987 std = 0.010
```

The core of the script is the `cross_validation.cross_val_score` function. The function in our script receives the following parameters:

- A learning algorithm (`estimator`)
- A training set of predictors (`X`)
- A target variable (`y`)
- The number of cross-validation folds (`cv`)
- A scoring function (`scoring`)
- The number of CPUs to be used (`n_jobs`)

Given such input, the function wraps some complex functions. It creates n-iterations, training a model of the n-cross-validation in-samples and testing and storing its score on the out-of-sample fold at each iteration. In the end, the function reports a list of the recorded scores of this kind:

```
In: scores
Out: array([ 0.96899225, 0.96899225, 0.9921875, 0.98412698,
0.99206349, 1, 1., 0.984, 0.99186992, 0.98347107])
```

The main advantage of using `cross_val_score` resides in its simplicity of usage and the fact that it automatically incorporates all the necessary steps for a correct cross-validation. For example, when deciding how to split the train sample into folds, if a y vector is provided, it keeps the same target class label's proportion in each fold as it was in the y provided.

Using cross-validation iterators

Though the `cross_val_score` function from the `cross_validation` module acts as a complete helper function for most of the cross-validation purposes, you may have the necessity to build up your own cross-validation processes. In this case, the same `cross_validation` module provides you with a formidable selection of iterators.

Before examining the most useful ones, let's provide a clear overview of how they all work by studying how one of the iterators, `cross_validation.KFold`, works.

`KFold` is quite simple in its functionality. If n-number of folds are given, it returns n iterations to the indexes of the training and validation sets for the testing of each fold.

Let's say that we have a training set made up of 100 examples and we would like to create a 10-fold cross-validation. First of all, let's set up our iterator:

```
In: kfolding = cross_validation.KFold(n=100, n_folds=10, shuffle=True, random_state=1)
for train_idx, validation_idx in kfolding:
    print train_idx, validation_idx
Out: [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 34 35 37 38 39 40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 66 67 68 70 71 72 73 74
75 76 77 78 79 83 85 86 87 88 89 90 91 92 94 95 96 97 98 99] [17 33
36 65 69 80 81 82 84 93]
...
```

By using the n parameter, we can instruct the iterator to perform the folding on 100 indexes. The `n_folds` specifies the number of folds. While the shuffle is set to `True`, it will randomly choose the fold components. Instead, if it is set to `false`, the folds will be created with respect to the order of the indexes (so, the first fold will be `[0 1 2 3 4 5 6 7 8 9]`).

As usual, the `random_state` parameter allows reproducibility of the folds generation.

During the iterator loop, the indexes for training and validation are provided with respect to your hypothesis for evaluation (let's figure out the h1, the linear SVC). You just have to select both `X` and `y` accordingly with the help of fancy indexing:

```
In: h1.fit(X[train_idx],y[train_idx])
h1.score(X[validation_idx],y[validation_idx])
Out: 0.90000000000000002
```

As you can see, a cross-validation iterator provides you with just the index functionality, and it is up to you when it comes to using indexes for your scoring evaluation on your hypothesis. This opens up for you opportunities for elaborate and sophisticated operations.

Among the other most useful, iterators, the following are worth a mention:

- `StratifiedKFold` works like `Kfold`, but it always returns folds with approximately the same class percentage as the training set. Instead of the number of cases, as an input parameter, it needs the target variable y. It is actually the iterator wrapped by default in the `cross_val_score` function that was just seen in the preceding section.

- LeaveOneOut works like Kfold, but it returns as a validation set only one observation. So in the end, the number of folds will be equivalent to the number of examples in the training set. We recommend that you use this cross-validation approach only when the training set is small, especially if there are less than 100 observations and a k-fold validation would reduce the training set a lot.
- LeavePOut is similar in advantages and limitations to LeaveOneOut, but its validation set is made up of P cases. So, the number of total folds will be the combination of P cases from all the available cases (which actually could be quite a large number as the size of your dataset grows).
- LeaveOneLabelOut provides a convenient way to cross-validate according to a scheme that you have prepared or computed in advance. In fact, it will act like Kfolds but for the fact that the folds will already be labeled and provided to the labels parameter.
- LeavePLabelOut is a variant of LeaveOneLabelOut. In this instance, the test folds are made of a number P of labels according to the scheme that you prepare in advance.

> To know more about the specific parameters required by each iterator, we suggest that you check out the Scikit-learn website, http://scikit-learn.org/stable/modules/classes.html#module-sklearn.cross_validation.

Sampling and bootstrapping

After illustrating iterators based on folds, p-out, and custom schemes, we'll continue our excursus on cross-validation iterators and quote all the sampling-based ones.

The sampling schemes are different because they do not split the training set, but they subsample or bootstrap it.

Subsampling is performed when you randomly select a part of the available data, obtaining a smaller dataset than the initial one.

Subsampling is very useful especially when you need to extensively test your hypothesis but you prefer not to obtain your validation from extremely small test samples (so, you can opt out of a leave-one-out approach or a KFold using a large number of folds). The following is an example of the same.

```
In: subsampling = cross_validation.ShuffleSplit(n=100, n_iter=10, 
test_size=0.1, random_state=1)
```

```
for train_idx, validation_idx in subsampling:
    print train_idx, validation_idx
Out:
[92 39 56 52 51 32 31 44 78 10  2 73 97 62 19 35 94 27 46 38 67 99 54
95 88 40 48 59 23 34 86 53 77 15 83 41 45 91 26 98 43 55 24  4 58 49
21 87  3 74 30 66 70 42 47 89  8 60  0 90 57 22 61 63  7 96 13 68 85
14 29 28 11 18 20 50 25  6 71 76  1 16 64 79  5 75  9 72 12 37] [80
84 33 81 93 17 36 82 69 65]
...
```

Similar to the other iterators, `n_iter` will set the number of subsamples `test_size` the percentage (if a float is given) or the number of observations to be used as a test.

Bootstrap, as a resampling method, has been used for a long time to estimate the sampling distribution of statistics. So, it is a proper method according to the evaluation of the out-of-sample performance of a machine learning hypothesis.

It works randomly—choosing observations and allowing repetitions—until a new dataset, which is of the same size as the original one, is built.

Unfortunately, since bootstrapping works by sampling with the replacement (that is, by allowing the repetition of the same observation), there are issues that arise due to the following:

- Cases that may appear both on the training and the test set (you just have to use out-of-bootstrap sample observations for test purposes)
- There is less variance and more bias than the cross-validation estimations due to nondistinct observations resulting from sampling with replacement

Scikit-learn's `cross_validation` module offered a bootstrap iterator. Anyway, the `Bootstrap` function has been deprecated as of version 0.15 and it will be removed in 0.17 because it implements a nonstandard way to perform cross-validation (it first separated the dataset in the train and test set and then bootstrapped each of them).

Although the function is useful (at least from our point of view as data science practitioners), we propose to you a simple replacement for `Bootstrap` that is suitable for cross-validating and which can be called by a for loop. It generates a sample bootstrap of the same size as the input data (the length of the indexes) and a list of the excluded indexes (out of the sample) that could be used for testing purposes:

```
In: import random
def Bootstrap(n, n_iter=3, random_state=None):
    """
    Random sampling with replacement cross-validation generator.
    For each iter a sample bootstrap of the indexes [0, n) is
    generated and the function returns the obtained sample
    and a list of all the excluded indexes.
    """
    if random_state:
        random.seed(random_state)
    for j in range(n_iter):
        bs = [random.randint(0, n-1) for i in range(n)]
        out_bs = list({i for i in range(n)} - set(bs))
        yield bs, out_bs

boot = Bootstrap(n=100, n_iter=10, random_state=1)
for train_idx, validation_idx in boot:
    print train_idx, validation_idx

Out:
[37, 12, 72, 9, 75, 5, 79, 64, 16, 1, 76, 71, 6, 25, 50, 20, 18, 84,
11, 28, 29, 14, 50, 68, 87, 87, 94, 96, 86, 13, 9, 7, 63, 61, 22, 57,
1, 0, 60, 81, 8, 88, 13, 47, 72, 30, 71, 3, 70, 21, 49, 57, 3, 68,
24, 43, 76, 26, 52, 80, 41, 82, 15, 64, 68, 25, 98, 87, 7, 26, 25,
22, 9, 67, 23, 27, 37, 57, 83, 38, 8, 32, 34, 10, 23, 15, 87, 25, 71,
92, 74, 62, 46, 32, 88, 23, 55, 65, 77, 3] [2, 4, 17, 19, 31, 33, 35,
36, 39, 40, 42, 44, 45, 48, 51, 53, 54, 56, 58, 59, 66, 69, 73, 78,
85, 89, 90, 91, 93, 95, 97, 99]
…
```

The function performs subsampling and accepts the parameter `n` for the `n_iter` index to draw the bootstrap samples and the `random_state` index for repeatability.

Hyper-parameters' optimization

A machine learning hypothesis is not only determined by the learning algorithm, but also by its hyper-parameters (the parameters of the algorithm that have to be a priori fixed and which cannot be learned during the training process) and the selection of variables to be used to achieve the best learned parameters.

In this section, we will explore how to extend the cross-validation approach to find the best hyper-parameters that are able to generalize to our test set. We will keep on using the handwritten digits dataset offered by the Scikit-learn package. Here's a useful reminder about how to load the dataset:

```
In: from sklearn.datasets import load_digits
digits = load_digits()
X, y = digits.data, digits.target
```

Also, we will keep on using support vector machines as our learning algorithm:

```
In: from sklearn import svm
h = svm.SVC()
hp = svm.SVC(probability=True, random_state=1)
```

This time, we will use two hypotheses. The first hypothesis is just the plain SVC that just guesses a label. The second hypothesis is SVC with the computation of label probabilities (parameter `probability=True`) with the `random_state` fixed to `1` for the reproducibility of the results. SVC is useful for all the loss metrics that require a probability and not a prediction, such as AUC, to evaluate the machine learning estimator's performance.

After doing this, we are ready to import the `grid_search` module and set the list of hyper-parameters that we want to test by cross-validation.

We are going to use the `GridSearchCV` function, which will automatically search for the best parameters according to a search schedule and score the results with respect to a predefined or custom scoring function:

```
In: from sklearn import grid_search
search_grid = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel':
['rbf']},
 ]
scorer = 'accuracy'
```

Now, we have imported the module, set the scorer variable using a string (`accuracy`), and created a list made of two dictionaries.

The scorer is a string that we chose from a range of possible ones that you can find in the 3.5.1.1. Common cases: predefined values section of the Scikit-learn documentation, which can be viewed at `http://scikit-learn.org/stable/modules/model_evaluation.html`.

Using predefined values just requires you to pick your evaluation metric from the list (there are some for classification and regression, and there are also some for clustering) and use the string by plugging it directly, or by using a string variable, into the `GridSearchCV` function.

`GridSearchCV` also accepts a parameter called `param_grid`, which can be a dictionary containing, as keys, an indication of all the hyper-parameters to be changed and, as values of the dictionary keys, lists of parameters to be tested. So, if you want to test the performances of your hypothesis with respect to the hyper-parameter C, you can create a dictionary like this:

```
{'C' : [1, 10, 100, 1000]}
```

Alternatively, according to your preference, you can use a specialized NumPy function to generate numbers that are evenly spaced on a log scale (as we have seen in the previous chapter):

```
{'C' :np.logspace(start=-2, stop=3, num=6, base=10.0)}
```

You can therefore enumerate all the possible parameters' values and test all their combinations. However, you can also stack different dictionaries, with each dictionary containing only a portion of the parameters that can be tested together. For example, when working with SVC, the kernel set to *linear* automatically excludes the gamma parameter. Combining it with the linear kernel would be in fact a waste of computational power since it would not have any effect on the learning process.

Now, let's proceed with the grid search, timing it (thanks to the `%timeit` IPython magic command) to know how much time it will take to complete the entire procedure:

```
In: search_func = grid_search.GridSearchCV(estimator=h,
param_grid=search_grid, scoring=scorer, n_jobs=-1, iid=False,
refit=True, cv=10)
%timeit search_func.fit(X,y)
print search_func.best_estimator_
print search_func.best_params_
print search_func.best_score_

Out: 1 loops, best of 3: 9.56 s per loop
SVC(C=10, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.001,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
{'kernel': 'rbf', 'C': 10, 'gamma': 0.001}
0.981081122784
```

It took about 10 seconds to complete the search on our computer. The search pointed out that the best solution is a SVC with rbf kernel, C=10, and gamma=0.001 with a cross-validated mean accuracy of 0.981.

As for the GridSearchCV command, apart from our hypothesis (the estimator parameter), param_grid, and the scoring we just talked about, we decided to set other optional but useful parameters:

1. First of all, we will set n_jobs=-1. This forces the function to use all the processors available on the computer, we run the IPython cell.
2. We will then set refit=True so that the function fits the whole training set using the best estimator's parameters. Now, we just need to apply the search_funct.predict() method to fresh data in order to obtain new predictions.
3. The cv parameter is set to 10 folds (however, you can go for a smaller number, trading off speed with accuracy of testing).
4. The iid parameter is set to False. This parameter decides how to compute the error measure with respect to the classes. If the classes are balanced (as in this case), setting iid won't have much effect. However if they are unbalanced, by default, iid=True will make the classes with more examples weigh more on the global error. Instead, iid=False means that all the classes should be considered the same. Since we wanted SVC to recognize every handwritten number from 0 to 9 no matter how many examples were given for each of them, we decided that setting the iid parameter to False was the right choice. According to your data science project, you may decide that you actually prefer the default set to True.

Building custom scoring functions

For our experiment, we picked a predefined scorer function. For classification, there are six measures available, and for regression, there are three. Though they are some of the most common measures, you may have to use a different measure. In our example, we may find it useful to use a loss function in order to figure out if, even when the classifier is wrong, the right answer is still ranked high in probability (so, the right answer is the second or the third option of the algorithm). How do we manage that?

In the sklearn.metrics module, there's actually a log_loss function. All that we have to do is to wrap it in a way that GridSearchCV might use it:

```
In: from sklearn.metrics import log_loss, make_scorer

Log_Loss = make_scorer(log_loss, greater_is_better=False,
needs_proba=True)
```

Here it is. Basically, a one-liner. We created another function (`Log_Loss`) by calling `make_scorer` to the `log_loss` error function from `sklearn.metrics`. We also want to point out that we want to minimize this measure (it is a loss, not a score) by setting `greater_is_better=False`. We will also specify that it works with probabilities, not predictions (so, set `needs_proba=True`). Since it works with probabilities, we will use the hp hypothesis, which was just defined in the preceding section, since SVC otherwise won't emit any probability for its predictions:

```
In: search_func = grid_search.GridSearchCV(estimator=hp,
param_grid=search_grid, scoring=Log_Loss, n_jobs=-1, iid=False,
refit=True, cv=3)
search_func.fit(X,y)
print search_func.best_score_
print search_func.best_params_
Out: -0.16138394082
{'kernel': 'rbf', 'C': 1, 'gamma': 0.001}
```

Now, our hyper-parameters are optimized for log loss, not for accuracy.

> A nice thing to remember—optimizing for the right function can bring much better results to your project. So, time spent working on the score function is always time well spent in data science.

At this point, let's imagine that you have a challenging task. Since it is easy to mistaken handwritten numbers 1 and 7, you have to optimize your algorithm to minimize its mistakes on these two numbers.

Ready? You can do this by defining a new loss function:

```
In: import numpy as np
from sklearn.preprocessing import LabelBinarizer
def my_custom_log_loss_func(ground_truth, p_predictions, penalty =
list(), eps=1e-15):
    adj_p = np.clip(p_predictions, eps, 1 - eps)
    lb = LabelBinarizer()
    g = lb.fit_transform(ground_truth)
    if g.shape[1] == 1:
        g = np.append(1 - g, g, axis=1)
    if penalty:
        g[:,penalty] = g[:,penalty] * 2
    summation = np.sum(g * np.log(adj_p))
    return summation * (-1.0/len(ground_truth))
```

As a general rule, the first parameter of your function should be the actual answer and the second should be the predictions or the predicted probabilities. You can also add parameters that have a default value or which allow you to have their values fixed later on when you call the `make_scorer` function:

```
In: my_custom_scorer = make_scorer(my_custom_log_loss_func,
greater_is_better=False, needs_proba=True, penalty = [4,9])
```

In this case, we set the penalty on for highly confusable numbers 4 and 9 (however, you can change it or even leave it empty to check whether the resulting loss will be the same as that of the previous experiment with the `sklearn.metrics.log_loss` function).

Now, this new loss function will double `log_loss` when evaluating the results of the classes of numbers 4 and 9:

```
In: from sklearn import grid_search


search_grid = [{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
{'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']}]


search_func = grid_search.GridSearchCV(estimator=hp, param_grid=search_
grid, scoring=my_custom_scorer, n_jobs=1,
iid=False, refit=True, cv=3)
search_func.fit(X,y)
print search_func.best_score_
print search_func.best_params_
Out: -0.199610271298
{'kernel': 'rbf', 'C': 1, 'gamma': 0.001}
```

> Please note that for the last example, we set `n_jobs=1`. There's a technical reason behind the same. If you are running this code on Windows (or the Unix system, it is fine), you may incur an error that may block your IPython notebook. All cross-validation functions (and many others) on Scikit-learn package work by using multiprocessors thanks to the Joblib package that requires all the functions to be run on multiple processors to be imported, and not defined (they should be pickable). A possible workaround is the saving of the function in a file on the disk, such as `custom_measure.py`, and importing it with the `from custom_measure import Log_Loss` command.

Reducing the grid search runtime

The GridSearchCV function can really do extensive work for you by checking all combinations of parameters, as required by your grid specification. Anyway, when the data or grid search space is big, the procedure may take a long time to compute.

As a different approach, the grid_search module offers RandomizedSearchCV, a procedure that randomly draws a sample of combinations and reports the best combination found.

This has some clear advantages:

- You can limit the number of computations.
- You can obtain a good result or, at worst, understand where to focus your efforts on in the grid search.

  RandomizedSearchCV has the same options as GridSearchCV but:

- An n_iter parameter, which is the number of random samples.
- A param_distributions, which has the same function as that of param_grid. However, it accepts only dictionaries and it works even better if you assign distributions as values, and not lists of discrete values. For instance, instead of C: [1, 10, 100, 1000], you can assign a distribution such as C:scipy.stats.expon(scale=100).

Let's test this function with our previous settings:

```
In: search_dict = {'kernel': ['linear','rbf'],'C': [1, 10, 100, 
1000], 'gamma': [0.001, 0.0001]}
scorer = 'accuracy'
search_func = grid_search.RandomizedSearchCV(estimator=h, 
param_distributions=search_dict, n_iter=7, scoring=scorer, 
                 n_jobs=-1, iid=False, refit=True, cv=10)
%timeit search_func.fit(X,y)
print search_func.best_estimator_
print search_func.best_params_
print search_func.best_score_
Out: 1 loops, best of 3: 6.19 s per loop
SVC(C=1000, cache_size=200, class_weight=None, coef0=0.0, degree=3,
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False)
{'kernel': 'rbf', 'C': 1000, 'gamma': 0.001}
0.981081122784
```

Using just half of the computations (7 draws against 14 trials with the exhaustive grid search), it found an equivalent solution. Let's also have a look at the combinations that have been tested:

```
In: search_func.grid_scores_
Out: [mean: 0.96108, std: 0.02191, params: {'kernel': 'linear', 'C':
1, 'gamma': 0.0001},
 mean: 0.98108, std: 0.01551, params: {'kernel': 'rbf', 'C': 1000,
'gamma': 0.001},
 mean: 0.97164, std: 0.02044, params: {'kernel': 'rbf', 'C': 100,
'gamma': 0.0001},
 mean: 0.96108, std: 0.02191, params: {'kernel': 'linear', 'C': 1000,
'gamma': 0.0001},
 mean: 0.96108, std: 0.02191, params: {'kernel': 'linear', 'C': 10,
'gamma': 0.0001},
 mean: 0.96108, std: 0.02191, params: {'kernel': 'linear', 'C': 1,
'gamma': 0.0001},
 mean: 0.96108, std: 0.02191, params: {'kernel': 'linear', 'C': 10,
'gamma': 0.0001}]
```

Even without a complete overview of all combinations, a good sample can prompt you to look for just the RBF kernel and for certain C and gamma ranges, limiting a following grid search to a limited portion of the potential search space.

Feature selection

With respect to the machine learning algorithm that you are going to use, irrelevant and redundant features may play a role in the lack of interpretability of the resulting model, long training times and, most importantly, overfitting and poor generalization.

Overfitting is related to the ratio of the number of observations and the variables available in your dataset. When the variables are many compared to the observations, your learning algorithm will have more chance of ending up with some local optimization or the fitting of some spurious noise due to the correlation between variables.

Apart from dimensionality reduction, which requires you to transform data, feature selection can be the solution to the aforementioned problems. It simplifies high dimensional structures by choosing the most predictive set of variables, that is, it picks the features that work well together, even if some of them are not such good predictors on an independent level.

The Scikit-learn package offers a wide range of feature selection methods:

- Univariate selection
- Recursive elimination
- Randomized Logistic Regression/stability selection
- L1-based feature selection
- Tree-based feature selection

Univariate and recursive elimination can be found in the `feature_selection` module. The others are a by-product of specific machine learning algorithms. Apart from tree-based selection (which will be mentioned in *Chapter 4*, *Machine Learning*), we are going to present all the methods and point out how they can help you improve your learning from the data.

Univariate selection

With the help of univariate selection, we intend to select single variables that are associated the most with your target variable according to a statistical test.

There are three available tests:

- The `f_regression` object uses an F-test and a p-value according to the ratio of explained variance against the unexplained one in a linear regression of the variable with the target. This is useful only for regression problems.
- The `f_classif` object is an Anova F test that can be used when dealing with classification problems.
- The `Chi2` object is a chi-squared test, which is suitable when the target is a classification and the variables are count or binary data (they should be positive).

All the tests have a score and a p-value. Higher scores and p-values indicate that the variable is associated and is therefore useful to the target. The tests do not take into account instances where the variable is a duplicate or is highly correlated to another variable. It is therefore mostly useful to rule out the not-so-useful variables than to highlight the most useful ones.

In order to automate the procedure, there are also some selection routines available:

- SelectKBest, based on the score of the test, takes the k best variables
- SelectPercentile, based on the score of the test, takes the top percentile of performing variables
- Based on the p-values of the tests, SelectFpr (false positive rate test), SelectFdr (false discovery rate test), and SelectFwe (family wise error rate procedure)

You can also create your own selection procedure with the GenericUnivariateSelect function using the score_func parameter, which takes predictors and the target and returns a score and a p-value based on your favorite statistical test.

The great advantage offered by these functions is that they offer a series of methods to select the variables (fit) and later on reduce (transform) all the sets to the best variables. In our example, we use the .get_support() method in order to get a Boolean indexing from both Chi2 and f_classif tests on the top 25 percent predictive variables. We then decide on the variables selected by both the tests:

```
In: from sklearn.datasets import make_classification
X, y = make_classification(n_samples=800, n_features=100,
n_informative=25, n_redundant=0, random_state=101)
```

make_classification creates a dataset of 800 cases and 100 features. The important variables are a quarter of the total:

```
In: from sklearn.feature_selection import SelectPercentile
from sklearn.feature_selection import chi2, f_classif
from sklearn.preprocessing import Binarizer, scale
Xbin = Binarizer().fit_transform(scale(X))
# if you use chi2, input X must be non-negative: X must contain
booleans or frequencies
# hence the choice to binarize after the normalization if the
variable if above the average
Selector_chi2 = SelectPercentile(chi2, percentile=25).fit(Xbin, y)
Selector_f_classif = SelectPercentile(f_classif,
percentile=25).fit(X, y)
chi_scores = Selector_chi2.get_support()
f_classif_scores = Selector_f_classif.get_support()
selected = chi_scores & f_classif_scores # use the bitwise and operator
```

The final selected variable contains a Boolean vector, pointing out 21 predictive variables that are pointed out by both the tests.

> As a suggestion based on experience, by operating with different statistical tests and retaining a high percentage of your variables, you can usefully exploit univariate selection by ruling out less informative variables and thus simplify your set of predictors.

Recursive elimination

The problem with univariate selection is the likelihood of selecting a subset containing redundant information, whereas our interest is to get a minimum set that works with our predictor algorithm. A recursive elimination in this case could help provide the answer.

By running the script that follows, you'll find the reproduction of a problem that is quite challenging and which you may also often come across in datasets of different cases and variable sizes:

```
In: from sklearn.cross_validation import train_test_split
X, y = make_classification(n_samples=100, n_features=100,
n_informative=5, n_redundant=2, random_state=101)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.30, random_state=101)


from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state=101)
classifier.fit(X_train, y_train)
print 'In-sample accuracy: %0.3f' % classifier.score(X_train,
y_train)
print 'Out-of-sample accuracy: %0.3f' % classifier.score(X_test,
y_test)
Out: In-sample accuracy: 1.000
Out-of-sample accuracy: 0.667
```

We have a small dataset with quite a large number of variables. It is a problem of the *p*>*n* type, where *p* is the number of variables and *n* is the number of observations.

In such cases, there are surely some informative variables in the dataset, but the noise provided by the others may fool the learning algorithm in assigning the correct coefficients to the correct features.

This reflects in high (in our case, perfect) in-sample accuracy. However, in a poorer test accuracy, the RFECV class, provided with a learning algorithm and instructions about the scoring/loss function and the cross-validation procedure, starts fitting an initial model on all variables and calculates a score based on cross-validation. At this point, RFECV starts pruning the variables until it reaches a set of variables where the cross-validated score starts decreasing (whereas, by pruning, the score should have stayed stable or increased):

```
In: from sklearn.feature_selection import RFECV
selector = RFECV(estimator=classifier, step=1, cv=10,
scoring='accuracy')
selector.fit(X_train, y_train)
print("Optimal number of features : %d" % selector.n_features_)
Out: Optimal number of features : 4
```

In our example, from 100 variables, the RFECV ended up selecting just four of them. We can check the result on the test set after transforming both the training and test set in order to reflect the variable pruning:

```
In: X_train_s = selector.transform(X_train)
X_test_s = selector.transform(X_test)
classifier.fit(X_train_s, y_train)
print 'Out-of-sample accuracy: %0.3f' % classifier.score(X_test_s,
y_test)
Out: Out-of-sample accuracy: 0.900
```

As a general rule, when you notice a large discrepancy between the training results (based on cross-validation, not the in-sample score) and the out-of-sample results, recursive selection can help you achieve better performance from your learning algorithms by pointing out some of the most important variables.

Stability and L1-based selection

However, though effective, recursive elimination is actually a greedy algorithm. While pruning, it opts for certain selections, potentially excluding many others. That's a good way to reduce an NP-hard problem, such as an exhaustive search among possible sets, into a more manageable one. Anyway, there's another way to solve the problem using all the variables at hand conjointly. Some algorithms use regularization to limit the weight of the coefficients, thus preventing overfitting and the selection of the most relevant variables without losing predictive power. In particular, the regularization L1 (the lasso) is well-known for the creation of sparse selections of variables' coefficients since it pushes many variables to the 0 value according to the set strength of regularization.

An example will clarify the usage of the logistic regression classifier and the synthetic dataset that we used for recursive elimination.

By the way, `linear_model.Lasso` will work out the L1 regularization for regression, whereas `linear_model.LogisticRegression` and `svm.LinearSVC` will do it for the classification:

```
In: from sklearn.svm import LinearSVC
classifier = LogisticRegression(C=0.1, penalty='l1', random_state=101) #
the smaller C the fewer features selected
classifier.fit(X_train, y_train)
print 'Out-of-sample accuracy: %0.3f' % classifier.score(X_test,
y_test)
Out: Out-of-sample accuracy: 0.933
```

The out-of-sample accuracy is better than the previous one that was obtained by using the greedy approach. The secret is the `penalty='l1'` and the `C` value that was assigned when initializing the `LogisticRegression` class. Since `C` is the main ingredient of the L1-based selection, it is important to choose it correctly. This can be done by using cross-validation, but there's an easier and an even more effective way: stability selection.

Stability selection uses L1 regularization even under the default values (though you may change them in order to improve the results) because it verifies its results by subsampling, that is, by recalculating the regularization process a large number of times using a randomly chosen part of the training dataset. The final result excludes all the variables that often had their coefficient estimated to zero. Only if a variable has most of the time a non zero coefficient will the variable be considered stable to the dataset and feature set variations, and important to be included in the model (hence the name stability selection).

Let's test this by implementing the selection approach (by using the dataset that we used before):

```
In: from sklearn.linear_model import RandomizedLogisticRegression
selector = RandomizedLogisticRegression(n_resampling=300,
random_state=101)
selector.fit(X_train, y_train)
print 'Variables selected: %i' % sum(selector.get_support()!=0)
X_train_s = selector.transform(X_train)
X_test_s = selector.transform(X_test)
classifier.fit(X_train_s, y_train)
```

```
print 'Out-of-sample accuracy: %0.3f' % classifier.score(X_test_s,
y_test)
Out: Variables selected: 3
Out-of-sample accuracy: 0.933
```

As a matter of fact, we obtained results that were similar to that of the L1-based selection by just using the default parameters of the `RandomizedLogisticRegression` class.

The algorithm works fine. It is reliable and out of the box (there are no parameters to tweak unless you want to try lowering the C values in order to speed it up). We suggest that you set the `n_resampling` parameter to a large number so that your computer can handle in a reasonable amount of time.

If you want to select for a regression problem, you should use the `RandomizedLasso` class instead.

Summary

In this chapter, we extracted significant meaning from data by applying a number of advanced data operations—from EDA and feature creation to dimensionality reduction and outlier detection.

More importantly, we started developing, with the help of many examples, our data science pipeline. This was achieved by encapsulating into a train/cross-validation/test setting our hypothesis that was expressed in terms of various activities—from data selection and transformation to the choice of the learning algorithm and its best hyper-parameters.

In the next chapter, we will delve into the principal machine learning algorithms offered by the Scikit-learn package, such as, among others, linear models, support vectors machines, ensembles of trees, and unsupervised techniques for clustering.

4
Machine Learning

After having illustrated all the data preparation steps in a data science project, we have finally arrived at the learning phase where algorithms are applied. In order to introduce you to the most effective machine learning tools that are readily available in Scikit-learn, we have prepared a brief introduction for all the major families of algorithms, complete with examples and tips on the hyper-parameters that guarantee the best possible results.

In this chapter, we will present the following topics:

- Linear and logistic regression
- Naive Bayes
- The k-Nearest Neighbors (kNN)
- Support Vector Machines (SVM)
- Ensembles such as Random Forests and Gradient Tree Boosting
- Stochastic gradient-based classification and regression for big data
- Unsupervised clustering with K-means and DBSCAN

Linear and logistic regression

Linear and logistic regressions are the two methods that can be used to linearly predict a target value and target class respectively. Let's start with an example of linear regression.

In this section, we will use the Boston dataset, which contains 506 samples, 13 features (all real numbers), and a (real) numerical target. We will divide our dataset into two sections by using a so-called train/test split cross-validation to test our methodology (in the example, 80 percent of our dataset goes in training, and 20 percent in the test):

```
In: from sklearn.datasets import load_boston
```

```
boston = load_boston()
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(boston.data,
boston.target, test_size=0.2, random_state=0)
```

The dataset is now loaded and the train/test pairs have been created. In the next few steps, we're going to train and fit the regressor in the training set and predict the target variable in the test dataset. We are then going to measure the accuracy of the regression task by using the MAE score (as explained in *Chapter 3, The Data Science Pipeline*). As for the scoring function, it's the Mean Absolute Error.

```
In: from sklearn.linear_model import LinearRegression
regr = LinearRegression()
regr.fit(X_train, y_train)
y_pred = regr.predict(X_test)
from sklearn.metrics import mean_absolute_error
print "MAE", mean_absolute_error(y_test, y_pred)
Out: MAE 3.84281058945
```

Great! We achieved our goal in the simplest possible way. Now, let's take a look at the time needed to train the system:

```
In: %timeit regr.fit(X_train, y_train)
Out: 1000 loops, best of 3: 381 µs per loop
```

That was really quick! Results, of course, are not that great (see the comparison with another regressor based on Random Forest in the IPython Notebook), but linear regression offers a very good tradeoff between the performance and speed of training and simplicity. Now, let's take a look under the hood. Why is it so fast but not that accurate? The answer is somewhat expected—this is so because it's a very simple linear method.

Now, let's dig into a mathematical explanation of this technique. Let's name $X(i)$ the *ith* sample (it is actually a row vector of numerical features) and $Y(i)$ its target. The goal of linear regression is to find a good weight (column) vector W, which is better at approximating the target value when multiplied by the observation vector, that is, $X(i) * W \approx Y(i)$ (note that this is a dot product). W should be the same, and the best, for every observation. Thus, solving the following equation becomes easy:

$$\begin{bmatrix} X(0) \\ X(1) \\ \vdots \\ X(n) \end{bmatrix} * \boldsymbol{W} = \begin{bmatrix} Y(0) \\ Y(1) \\ \vdots \\ Y(n) \end{bmatrix}$$

W can be easily found with the help of a matrix inversion (or, more likely, a pseudo-inversion, which is a computationally efficient way) and a dot product. Here's the reason why linear regression is so fast. Note that this is a simplistic explanation—the real method adds another virtual feature to compensate for the bias of the process. Yet, this does not change the complexity of the regression algorithm much.

We progress now to logistic regression. In spite of what the name suggests, it is a classifier and not a regressor. It must be used in classification problems where you are dealing with only two classes (binary classification). Typically, target labels are Boolean, that is ,they have values as either True/False or 0/1 (indicating the presence or absence of the expected outcome). In our example, we keep on using the same dataset. The target is to guess whether a house value is over or under the average of any other threshold value that we are interested in (that is, we moved from a regression problem to a binary classification one). We start preparing the dataset by using the following commands:

```
In: import numpy as np
avg_price_house = np.average(boston.target)
high_priced_idx = (y_train >= avg_price_house)
y_train[high_priced_idx] = 1
y_train[np.logical_not(high_priced_idx)] = 0
y_train = y_train.astype(np.int8)
high_priced_idx = (y_test >= avg_price_house)
y_test[high_priced_idx] = 1
y_test[np.logical_not(high_priced_idx)] = 0
y_test = y_test.astype(np.int8)
```

Now, we will train and apply the classifier. To measure its performance, we will simply print the classification report:

```
In: from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
from sklearn.metrics import classification_report
print classification_report(y_test, y_pred)
Out:
             precision    recall  f1-score   support
          0       0.81      0.90      0.85        61
          1       0.82      0.68      0.75        41
avg / total       0.83      0.81      0.81       102
```

The output of this command can change on your machine.

The `precision` and `recall` values are over `80` percent. This is already a good result for a very simple method. The training speed is impressive, too. Thanks to IPython, we can have a comparison of the algorithm with a more advanced classifier in terms of performance and speed:

```
In: %timeit clf.fit(X_train, y_train)
100 loops, best of 3: 2.54 ms per loop
```

What's under the hood of logistic regression? The simplest classifier a person could imagine is a linear regressor followed by a hard threshold:

$$y\_pred_i = sign\left(X_i * W\right)$$

Here, *sign(a) = +1* if *a* is greater or equal than zero, 0 otherwise.

To smooth down the hardness of the threshold and predict the probability of belonging to a class, logistic regression uses the `logit` function. Its output is a [0 to 1] real number, which indicates the probability that the observation belongs to class 1. With formulas, that becomes:

$$Prob\left(y_i = +1 \mid X_i\right) = logistic\left(X_i \cdot W\right)$$

Here, logistic(α) = $e^{\alpha} / \left(1 + e^{\alpha}\right)$.

> Why is the *logistic* function used instead of some other function? Well, it just works pretty well in most real cases. In the remaining cases, if you're not completely satisfied with it, you may want to try some other nonlinear functions instead.

Naive Bayes

Naive Bayes is a very common classifier used for probabilistic multiclass classification. Given the feature vector, it uses the Bayes rule to predict the probability of each class. It's often applied to text classification since it's very effective with large and fat data (with many features) with a consistent a priori probability.

There are three kinds of Naive Bayes classifiers; each of them has strong assumptions (hypotheses) about the features. If you're dealing with real/continuous data, the Gaussian Naive Bayes classifier assumes that features are generated from a Gaussian process (that is, they are normally distributed). Alternatively, if you're dealing with an event model where events can be modelled with a multinomial distribution (in this case, features are counters or frequencies), you need to use the Multinomial Naive Bayes classifier. Finally, if all your features are independent and Boolean, and it is safe to assume that they're the realization of a Bernullian process, you can use the Bernoulli Naive Bayes classifier.

Let's see an example of the application of the Gaussian Naive Bayes classifier. An example of text classification is given at the end of this chapter. You can try it by substituting the SGDClassifier of the example with a MultinomialNB. In the following example, we're going to use the Iris dataset, assuming that the features are Gaussians:

```
In: from sklearn import datasets
iris = datasets.load_iris()
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(iris.data,
iris.target, test_size=0.2, random_state=0)

In: from sklearn.naive_bayes import GaussianNB
clf = GaussianNB()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

In: from sklearn.metrics import classification_report
```

```
print classification_report(y_test, y_pred)
Out:
             precision    recall  f1-score   support
          0       1.00      1.00      1.00        11
          1       0.93      1.00      0.96        13
          2       1.00      0.83      0.91         6
avg / total       0.97      0.97      0.97        30

In: %timeit clf.fit(X_train, y_train)
Out: 1000 loops, best of 3: 338 µs per loop
```

It seems to have a good performance and high training speed, although we shouldn't forget that our dataset is also very small. Now, let's see how it works on a multiclass problem.

The aim of the classifier is to predict the probability that a feature vector belongs to the *Ck* class. In the example, there are three classes [`setosa`, `versicolor`, and `virginica`]. So, we need to compute the membership probability for all classes. To make the explanation simple, let's name them *1*, *2*, and *3*. Therefore, the goal of the Naive Bayes classifier for the *i*th observation is to compute the following:

$$Prob\left(Ck \mid X\left(i\right)\right)$$

Here, *X(i)* is the vector of the features (in the example, it is composed of four real numbers), whose components are [*X(i, 0)*, *X(i, 1)*, *X(i, 2)*, *X(i, 3)*].

Using the Bayes' rule, it becomes:

$$Prob\left(Ck \mid X\left(i\right)\right) = \frac{Prob\left(Ck\right) \;\; Prob\left(X\left(i\right) \mid Ck\right)}{Prob\left(X\left(i\right)\right)}$$

We can describe the same formula, as follows:

The a-posteriori probability is the a-priori probability of the class multiplied by the likelihood and then divided by the evidence

From the probability theory, we know that the joint probability can be expressed as follows (simplifying the problem):

$$Prob\left(Ck, X\left(i,0\right),\ldots,X\left(i,n\right)\right) = Prob\left(X\left(i,0\right),\ldots,X\left(i,n\right) \mid Ck\right)$$

Then, the second factor of the multiplication can be rewritten as follows (conditional probability):

$$Prob\left(X\left(i,0\right) \mid Ck\right) \quad Prob\left(X\left(x,1\right),\ldots,X\left(i,n\right) \mid Ck, X\left(i,0\right)\right)$$

You can then use the conditional probability definition to express the second member of the multiplication. In the end, you'll have a very long multiplication:

$$\begin{gathered} Prob\left(Ck, X\left(i,0\right),\ldots,X\left(i,n\right)\right) = \\ \ldots \\ Prob\left(Ck\right) \quad Prob\left(X\left(i,0\right) \mid Ck\right) \quad Prob\left(X\left(i,1\right) \mid Ck, X\left(i,0\right)\right)\ldots \end{gathered}$$

The naive assumption is that each feature is considered conditionally independent of the other features for each class. So, the probabilities can be simply multiplied. The formula for the same is as follows:

$$Prob\left(X\left(i,0\right) \mid Ck, X\left(:\right)\right) = Prob\left(X\left(i,0\right) \mid Ck\right)$$

Therefore, wrapping up the math, to select the best class, the following formula is used:

$$Y_{pred}\left(i\right) = \underset{k=0,1,2,3}{argmax} \quad Prob\left(Ck\right)\prod_{k=0}^{n-1} Prob\left(X\left(i,k\right) \mid Ck\right)$$

That is a simplification, since the evidence probability (the denominator of the Bayes' rule) has been removed, and since all the classes would have the same division.

From the previous formula, you can understand why the learning phase is so quick—it's just a count of occurrences.

Note that for this classifier, a corresponding regressor doesn't exist.

The k-Nearest Neighbors

K-Nearest Neighbors, or simply kNN, belongs to the class of instance-based learning, also known as lazy classifiers. It's one of the simplest classification methods because the classification is done by just looking at the K closest examples in the training set (in terms of Euclidean distance or some other kind of distance) in the case that we want to classify. Then, given the K similar examples, the most popular target (majority voting) is chosen as the classification label. Two parameters are mandatory for this algorithm: the neighborhood cardinality (K) and the measure to evaluate the similarity (although the Euclidean distance, or L2, is the most used and is the default parameter for most implementations).

Let's take a look at an example. We are going to use a pretty big dataset, the `MNIST` handwritten digits. We will later explain why we use this dataset. We're going to use only a small portion of it (1000 samples) to keep the computational time reasonable and we shuffle the observations for obtaining better results (though as a consequence, your final output may be slightly different from ours).

```
In: from sklearn.utils import shuffle
from sklearn.datasets import fetch_mldata
from sklearn.cross_validation import train_test_split

mnist = fetch_mldata("MNIST original")
mnist.data, mnist.target = shuffle(mnist.data, mnist.target)

# We reduce the dataset size, otherwise it'll take too much time to run
mnist.data = mnist.data[:1000]
mnist.target = mnist.target[:1000]

X_train, X_test, y_train, y_test = train_test_split(mnist.data, mnist.
target, test_size=0.8, random_state=0)

In: from sklearn.neighbors import KNeighborsClassifier
# KNN: K=10, default measure of distance (euclidean)
clf = KNeighborsClassifier(3)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

In: from sklearn.metrics import classification_report
```

```
print classification_report(y_test, y_pred)
Out:
             precision    recall  f1-score   support
        0.0       0.68      0.90      0.78        79
        1.0       0.66      1.00      0.79        95
        2.0       0.83      0.50      0.62        76
        3.0       0.59      0.64      0.61        85
        4.0       0.65      0.56      0.60        75
        5.0       0.76      0.55      0.64        80
        6.0       0.89      0.69      0.77        70
        7.0       0.76      0.83      0.79        76
        8.0       0.91      0.56      0.69        77
        9.0       0.61      0.75      0.67        87
avg / total       0.73      0.70      0.70       800
```

The performance is not so great on this dataset, but remember that the classifier has to work on ten different classes. Now, let's check the time the classifier needs for the training and predicting:

```
In: %timeit clf.fit(X_train, y_train)
Out: 1000 loops, best of 3: 1.66 ms per loop

In: %timeit clf.predict(X_test)
Out: 10 loops, best of 3: 177 ms per loop
```

The training speed is exceptional. But now, think about the algorithm. The training phase is just copying the data and nothing else! In fact, the prediction speed is connected to the number of samples you have in your training step and the number of features composing it (that's actually the feature matrix number of elements). In all the other algorithms that we've seen, the prediction speed is independent of the number of training cases that we have in our dataset. In conclusion, we can say that kNN is great for small datasets, but it's definitely not the algorithm you would use when dealing with big data.

Just one last remark about this classification algorithm—you can also find the analogous regressor, `KNeighborsRegressor`, connected to it. Its algorithm is pretty much the same, except that the predicted value is the average of the K target values of the neighborhood.

Advanced nonlinear algorithms

Support Vector Machine (**SVM**) is a powerful and advanced supervised learning technique for classification and regression that can automatically fit linear and nonlinear models. Scikit-learn offers an implementation based on LIBSVM, a complete library of SVM classification and regression implementations, and LIBLINEAR, a library more scalable for linear classification of large datasets, especially the sparse text based ones. Both libraries have been developed at the National Taiwan University, and both have been written in C++ with a C API to interface with other languages. Both libraries have been extensively tested (being free, they have been used in other open source machine learning toolkits) and have been proven to be both fast and reliable. The C API explains well two tricky needs for them to operate better under the Python Scikit-learn:

- LIBSVM, when operating, needs to reserve some memory for kernel operations. The `cache_size` parameter is used to set the size of the kernel cache, which is specified in megabytes. Though the default value is 200, it is advisable to raise it to 500 or 1000, depending on your resources.
- They both expect C-ordered NumPy `ndarray` or SciPy `sparse.csr_matrix` (a row optimized sparse matrix kind), preferably with float64 dtype. If the Python wrapper receives them under a different data structure, it will have to copy the data in a suitable format, slowing down the process and consuming more memory.

SVM for classification

As an example for classification using SVM, we will use SVC with both a linear and an **RBF** kernel (RBF stands for **Radial Basis Function**, which is an effective nonlinear function). LinearSVC will be instead employed for a complex problem with a large number of observations (standard SVC won't perform with more than 10,000 observations; due to the growing cubic complexity, LinearSVC can instead scale linearly).

For our first classification example, a binary one, we'll take on a dataset from the IJCNN'01 neural network competition. It is a time series of 50,000 samples produced by a physical system of a 10-cylinder internal combustion engine. Our target is binary: normal engine firing or misfire. We will start retrieving the dataset directly from the LIBSVM web site; it is in the LIBSVM format and is compressed by Bzip2. So, we have to download it to our working directory:

```
In: import urllib2

target_page = 'http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
binary/ijcnn1.bz2'
```

```
with open('ijcnn1.bz2','wb') as W:
    W.write(urllib2.urlopen(target_page).read())

In: from sklearn.datasets import load_svmlight_file
X_train, y_train = load_svmlight_file('ijcnn1.bz2')
first_rows = 2500
X_train, y_train = X_train[:first_rows,:], y_train[:first_rows]
```

For exemplification purposes, we will limit the number of observations from 25,000 to 2,500. The number of available features is 22. Furthermore, we won't preprocess the data since it is already compatible with the SVM requirements of the rescaled features in the range of 0 and 1:

```
In: import numpy as np
from sklearn.cross_validation import cross_val_score
from sklearn.svm import SVC
hypothesis = SVC(kernel='rbf', degree=2, random_state=101)
scores = cross_val_score(hypothesis, X_train, y_train, cv=5,
scoring='accuracy', n_jobs=-1)
print "SVC with rbf kernel -> cross validation accuracy: mean = %0.3f
std = %0.3f" % (np.mean(scores), np.std(scores))
Out: SVC with rbf kernel -> cross validation accuracy: mean = 0.910
std = 0.001
```

In our example, we tried an SVC with an RBF kernel and a `degree` of `2` (it is a good solution but it is not optimal). All the other parameters were kept at the default values. You can try to modify `first_rows` to larger values (up to 25,000) and verify how the algorithm scales up to an increase in the number of observations. You will notice that the scaling is not linear, that is, the computation time will increase more than proportionally with the size of the data.

With regards to the SVM scalability, it is interesting to see how they behave with a multiclass problem and a large number of cases. In the Poker dataset (which can be found on the LIBSVM website, but is derived from the UCI repository – `archive.ics.uci.edu/ml/`), each record is an example of a poker hand consisting of five playing cards drawn from a standard deck of 52. Each card is described using two attributes (suit and rank) for a total of 10 predictive attributes. Since these two are not really metric measures, we need to hot-encode the features, that is, transform them into indicator features (1/0 binary variables). Our target is a multiclass attribute that we can call the *Poker Hand*:

- **0**: Nothing in hand; not a recognized poker hand
- **1**: One pair; one pair of equal ranks within five cards

- **2**: Two pairs; two pairs of equal ranks within five cards
- **3**: Three of a kind; three equal ranks within five cards
- **4**: Straight; five cards, sequentially ranked with no gaps
- **5**: Flush; five cards with the same suit
- **6**: Full house; pair + different rank that has three of a kind
- **7**: Four of a kind; four equal ranks within five cards
- **8**: Straight flush; straight + flush
- **9**: Royal flush; {Ace, King, Queen, Jack, Ten} + flush

Here is the script that you can use to load and prepare the dataset:

```
In: import urllib2
target_page = 'http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
multiclass/poker.bz2'
with open('poker.bz2','wb') as W:
    W.write(urllib2.urlopen(target_page).read())


In: import numpy as np
from sklearn.datasets import load_svmlight_file
X_train, y_train = load_svmlight_file('poker.bz2')
from sklearn.preprocessing import OneHotEncoder
hot_encoding = OneHotEncoder(sparse=True)
X_train = hot_encoding.fit_transform(X_train.toarray())
```

Our dataset is initially composed of 25,010 cases and 10 variables representing the 13 ranks and 4 suits of the five cards that are a part of a poker hand. The target variable is an ordinal ranging from 0 to 9, representing different poker combinations. Using `OneHotEncoder`, the initial 10 predictive variables are turned into 85 binary ones (because it is 5 times the sum of 13 ranks and 4 suits). If we consider that, since we have 10 classes, we will need to train 10 different classifiers focused on predicting a single class against the others. We will then have 2,50,100 data points for each cross-validation test. This is quite a challenge for many algorithms considering that there are 85 variables, but `LinearSVC` can handle it in a reasonable time:

```
In: from sklearn.cross_validation import cross_val_score
from sklearn.svm import LinearSVC
hypothesis = LinearSVC(dual=False)
```

```
scores = cross_val_score(hypothesis, X_train, y_train, cv=3,
scoring='accuracy', n_jobs=-1)

print "LinearSVC -> cross validation accuracy: mean = %0.3f std =
%0.3f" % (np.mean(scores), np.std(scores))

Out: LinearSVC -> cross validation accuracy: mean = 0.490 std =
0.004
```

The resulting accuracy is `0.49`, which is a good result. Yet, it surely leaves room for some further improvement. On the other hand, the problem seems to be a nonlinear one, though applying SVC with a nonlinear kernel would turn into a very long training process, as since the number of observations is large (from academic literature, we know that SVM should achieve a classification accuracy of around 0.60). We will reprise this problem in the following examples by using other nonlinear algorithms in order to check if we can improve the score obtained by LinearSVC.

SVM for regression

To provide an example on regression, we decided on a dataset of real estate prices of houses in California:

```
In: import urllib2

target_page =
'http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/regression/ca
data'

from sklearn.datasets import load_svmlight_file

X_train, y_train = load_svmlight_file(urllib2.urlopen(target_page))

from sklearn.preprocessing import scale

first_rows = 2000

X_train = scale(X_train[:first_rows,:].toarray())

y_train = y_train[:first_rows]/1000
```

The cases from the dataset are reduced to `2,000` for performance reasons. The features have been scaled in order to avoid the influence from the different scale of the original variables. Also, the target variable is divided by `1,000` in order to render it more readable in thousand dollar values.

```
In: from sklearn.cross_validation import cross_val_score

from sklearn.svm import SVR

hypothesis = SVR()

scores = cross_val_score(hypothesis, X_train, y_train, cv=3,
scoring='mean_absolute_error', n_jobs=-1)
```

```
print "SVR -> cross validation accuracy: mean = %0.3f std = %0.3f" %
(np.mean(scores), np.std(scores))
Out: SVR -> cross validation accuracy: mean = -57.838 std = 7.145
```

The chosen error is the mean absolute error, which is reported by the `sklearn` class as a negative number (but it is actually without a sign).

Tuning SVM

Before we start working on the hyper-parameters (which are typically a different set of parameters, depending on the implementation), there are two aspects that are left to be fixed.

The first is about the sensitivity of the SVM to variables of different scale and large numbers. Similar to other learning algorithms based on linear combinations, having variables at different scales leads the algorithm to be dominated by features with the larger range or variance. Moreover, extremely high or low numbers may cause problems to the optimization process of the learning algorithms. It is advisable to scale all the data to intervals such as [0,+1], which is a necessary choice if you are working with sparse arrays. In fact, it is desirable to preserve zero entries. Otherwise, data will become dense, consuming more memory. You can also scale the data to the [-1,+1] interval. Alternatively, you can standardize them to zero mean and unit variance. You can use, from the preprocessing module, the `MinMaxScaler` and `StandardScaler` utility classes by first fitting them on the training data and then transforming both the train and the test sets.

The second aspect is regarding unbalanced classes. The algorithm tends to favor the frequent classes more. A solution, apart from resampling, is to weight the C penalty parameter accordingly to the frequency of the class (low values will penalize the class more; high values less). There are two ways to do that in respect of the different implementations: the `class_weight` parameter in SVC (which can be set to the keyword `auto` or provided with specific values for each class in a dictionary) and `sample_weight` in the `.fit()` method of `SVC`, `NuSVC`, `SVR`, `NuSVR`, and `OneClassSVM` (it requires a unidimensional array, where each position refers to the weight of each training example).

Having dealt with scale and class balance, you can exhaustively search for optimal settings of the other parameters using `GridSearchCV` from the `grid_search` module in sklearn. Though SVM works fine with default parameters, they are often not optimal, and you do need to test by using cross-validation various value combinations in order to find the best ones.

According to their importance, you have to set the following parameters:

- `C`: This is the penalty value. Decreasing it increases penalization on noisy values. The suggested search range is `np.logspace(-3, 3, 7)`.
- `Kernel`: For this parameter, you should always use `linear` and `rbf`.
- `Degree`: With `rbf`, trying a sequence from 2 to 5 should suffice.
- `Gamma`: Only with an rbf kernel, high values tend to fit data in a better way. The suggested search range is `np.logspace(-3, 3, 7)`.
- `Nu`: For `NuSVC` and `NuSVR`, this parameter approximates the fraction of training errors and support vectors. It should be in the range of [0,1].
- `Epsilon`: This parameter specifies how many errors you are willing to accept for each training data observation. The suggested search range is `np.insert(np.logspace(-4, 2, 7, base=2),0, [0])`.
- The `loss, penalty`: For LinearSVC, this parameter accepts the (l1,l2),(l2,l1), and (l2,l2) combinations. The (l2,l1) combination practically transforms `LinearSVC` into `SVC(kernel='linear')`.

As an example, we will load the IJCNN'01 dataset again, and we will try to improve the initial accuracy of 0.91 by looking for a better degree, `C`, and `gamma` values. In order to save time, we will use the `RandomizedSearchCV` class to increase the accuracy to `0.998` (cross-validation estimate):

```
In: from sklearn.datasets import load_svmlight_file
X_train, y_train = load_svmlight_file('ijcnn1.bz2')
first_rows = 2500
X_train, y_train = X_train[:first_rows,:], y_train[:first_rows]
from sklearn.svm import SVC
from sklearn.grid_search import RandomizedSearchCV
hypothesis = SVC(kernel='rbf', random_state=101)
search_dict = {'degree':[2,3], 'C': [0.01, 0.1, 1, 10, 100, 1000],
'gamma': [0.1, 0.01, 0.001, 0.0001]}
search_func = RandomizedSearchCV(estimator=hypothesis,
param_distributions=search_dict, n_iter=30, scoring='accuracy',
  n_jobs=-1, iid=True, refit=True, cv=5, random_state=101)
search_func.fit(X_train, y_train)
print 'Best parameters %s' % search_func.best_params_
print 'Cross validation accuracy: mean = %0.3f' %
search_func.best_score_
Out: Best parameters {'C': 100, 'gamma': 0.1, 'degree': 3}
Cross validation accuracy: mean = 0.998
```

Ensemble strategies

Until now, we have seen single learning algorithms of growing complexity. Ensembles represent an effective alternative since they tend to achieve better predictive accuracy by combining or chaining the results from different data samples, algorithms settings, and types.

They divide themselves into two branches. According to the method used, they ensemble predictions:

- **Averaging algorithms**: These predict by averaging the results of various parallel estimators. The variations in the estimators provide further division into four families: pasting, bagging, subspaces, and patches.
- **Boosting algorithms**: These predict by using a weighted average of sequential aggregated estimators.

Before delving into some examples for both classification and regression, we will provide you with the steps to load a new dataset for classification, the Covertype dataset. It features a large number of 30×30 meter patches of forest in the US. The data pertaining to them is collected for the task of predicting the dominant species of tree of each patch (cover type). It is a multiclass classification problem (seven `covertypes` to predict). Each sample has 54 features, and there are over 5,80,000 samples (but for performance reasons, we will work with just 15,000 of these samples):

```
In: from sklearn.datasets import fetch_covtype

covertype_dataset = fetch_covtype(random_state=101, shuffle=True)

print covertype_dataset.DESCR

covertype_X = covertype_dataset.data[:15000,:]

covertype_y = covertype_dataset.target[:15000]

covertypes = ['Spruce/Fir', 'Lodgepole Pine', 'Ponderosa Pine',
'Cottonwood/Willow', 'Aspen', 'Douglas-fir', 'Krummholz']
```

Pasting by random samples

Pasting is the first type of averaging ensembling. In pasting, the estimators are built by using a large number of small samples from the data (sampling without replacement). It is very useful when dealing with very large data. It was devised by Leo Breiman, the creator of the RandomForest algorithm. There are no specific algorithms in the Scikit-learn leveraging pasting, though it is easily achievable using the sampling techniques that we illustrated in *Chapter 3*, *The Data Science Pipeline*.

Bagging with weak ensembles

Bagging works with samples in a way that is similar to that of pasting, but it allows replacement. Theoretically elaborated by Leo Breiman, bagging is implemented in a Scikit-learn class. You just have to decide the algorithm that you'd like to use for the training and plug it into BaggingClassifier, or BaggingRegressor for regression problems, and set a high number of estimators (and consequently, a high number of samples):

```
In: from sklearn.cross_validation import cross_val_score
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import fetch_covtype
import numpy as np
covertype_dataset = fetch_covtype(random_state=101, shuffle=True)
covertype_X = covertype_dataset.data[:15000,:]
covertype_y = covertype_dataset.target[:15000]
hypothesis = BaggingClassifier(KNeighborsClassifier(n_neighbors=1),
max_samples=0.7, max_features=0.7, n_estimators=100)
scores = cross_val_score(hypothesis, covertype_X, covertype_y, cv=3,
scoring='accuracy', n_jobs=-1)
print "BaggingClassifier -> cross validation accuracy: mean = %0.3f
std = %0.3f" % (np.mean(scores), np.std(scores))
Out: BaggingClassifier -> cross validation accuracy: mean = 0.793

std = 0.002
```

Good choices for the estimator are weak predictors. A weak learner in classification or prediction is just an algorithm which performs poorly just above the chance baseline. Some good examples are Naive Bayes and K Nearest Neighbors. The advantage of using weak learners and ensembling them is that they can be trained more quickly than complex algorithms. However, combined together, they can achieve comparable, or even better, predictive performances than more sophisticated single algorithms.

Random Subspaces and Random Patches

Random Subspaces is when estimators differentiate because of random subsets of the features. Instead, in Random Patches, estimators are built on subsets of both samples and features. Random Patches are implemented as `RandomForestClassifier` / `RandomForestRegressor` and `ExtraTreesClassifier` / `ExtraTreesRegressor`, which is a more randomized kind of RandomForests (there is a low variance in estimates, but a greater bias of estimators):

```
In: from sklearn.cross_validation import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import fetch_covtype
import numpy as np
covertype_dataset = fetch_covtype(random_state=101, shuffle=True)
covertype_X = covertype_dataset.data[:15000,:]
covertype_y = covertype_dataset.target[:15000]
hypothesis = RandomForestClassifier(n_estimators=100,
random_state=101)
scores = cross_val_score(hypothesis, covertype_X, covertype_y, cv=3,
scoring='accuracy', n_jobs=-1)
print "RandomForestClassifier -> cross validation accuracy:",
print "mean = %0.3f std = %0.3f" % (np.mean(scores), np.std(scores))
Out: RandomForestClassifier -> cross validation accuracy: mean =
0.808 std = 0.008
```

```
In: from sklearn.ensemble import ExtraTreesClassifier
hypothesis = ExtraTreesClassifier(n_estimators=100, random_state=101)
scores = cross_val_score(hypothesis, covertype_X, covertype_y, cv=3,
scoring='accuracy', n_jobs=-1)
print "ExtraTreesClassifier -> cross validation accuracy: mean =
%0.3f std = %0.3f" % (np.mean(scores), np.std(scores))
Out: ExtraTreesClassifier -> cross validation accuracy: mean = 0.821

std = 0.007
```

For both algorithms, the critical hyper-parameters that should be fixed are:

- `max_features`: This is the number of sampled features that are present at every split that can really determine the performance of the algorithm. The lower the number, the higher the bias.

- `min_samples_leaf` : This allows you to determine the depth of the trees. Large numbers diminish the variance and increases the bias.
- `bootstrap` : This is a Boolean which allows bootstrapping.
- `n_estimators` : This is the number of trees (remember that the more trees, the better, though that comes at a computational cost that you have to take into account).

Both RandomForests and ExtraTrees are really parallel algorithms. Don't forget to set the appropriate number of `n_jobs` to speed up the execution. When classifying, they decide for the most voted class (majority voting); when regressing, they simply average the resulting values. As an exemplification, we propose a regression example that is based on the California House prices dataset:

```
In: import urllib2
target_page =
'http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/regression/ca
data'
from sklearn.datasets import load_svmlight_file
X_train, y_train = load_svmlight_file(urllib2.urlopen(target_page))
from sklearn.preprocessing import scale
first_rows = 2000


In: X_train =X_train[:first_rows,:].toarray()
y_train = y_train[:first_rows]/1000.
from sklearn.ensemble import RandomForestRegressor
from sklearn.cross_validation import cross_val_score
import numpy as np
hypothesis = RandomForestRegressor(n_estimators=300,
random_state=101)
scores = cross_val_score(hypothesis, X_train, y_train, cv=3,
scoring='mean_absolute_error', n_jobs=-1)
print "RandomForestClassifier -> cross validation accuracy: mean =
%0.3f std = %0.3f" % (np.mean(scores), np.std(scores))
Out: RandomForestClassifier -> cross validation accuracy: mean = -46.646
std = 5.006
```

Sequences of models – AdaBoost

AdaBoost is another boosting algorithm. It fits a sequence of weak learners (originally stumps, that is, single-level decision trees) on reweighted versions of the data. Weights are assigned on the basis of the predictability of the case. More difficult cases are weighted more. The idea is that the trees first learn easy examples and then concentrate more on the difficult ones. In the end, the sequence of weak learners is weighted in order to maximize the result:

```
In: from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets import fetch_covtype
covertype_dataset = fetch_covtype(random_state=101, shuffle=True)
covertype_X, covertype_y = covertype_dataset.data[:15000,:],
covertype_dataset.target[:15000]
import numpy as np
hypothesis = AdaBoostClassifier(n_estimators=300, random_state=101)
scores = cross_val_score(hypothesis, covertype_X, covertype_y, cv=3,
scoring='accuracy', n_jobs=-1)
print "Adaboost -> cross validation accuracy: mean = %0.3f std =
%0.3f" % (np.mean(scores), np.std(scores))
Out: Adaboost -> cross validation accuracy: mean = 0.610 std = 0.014
```

Gradient tree boosting (GTB)

Gradient boosting is another version of boosting. Like AdaBoost, it is based on a gradient descent function. The algorithm has proved to be one of the most proficient ones from the ensemble, though it is characterized by major variance, more sensibility to noise in data (both problems could be attenuated by using subsampling), and increased computational cost due to nonparallel operations.

To demonstrate how GTB performs, we will again try using the problematic Poker dataset, which was already examined when illustrating SVM:

```
In: import urllib2
target_page = 'http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
multiclass/poker.bz2'
with open('poker.bz2','wb') as W:
    W.write(urllib2.urlopen(target_page).read())
from sklearn.datasets import load_svmlight_file
X_train, y_train = load_svmlight_file('poker.bz2')
```

```
from sklearn.preprocessing import OneHotEncoder
hot_encoding = OneHotEncoder(sparse=True)
X_train =
hot_encoding.fit_transform(X_train.toarray()).toarray()[:2500,:]
y_train = y_train[:2500]
```

After loading the data, the sample size is limited to 2500 observations in order to achieve a reasonable training performance:

```
In: from sklearn.ensemble import GradientBoostingClassifier
import numpy as np
hypothesis = GradientBoostingClassifier(max_depth=5,
n_estimators=300, random_state=101)
scores = cross_val_score(hypothesis, X_train, y_train, cv=3,
scoring='accuracy', n_jobs=-1)
print "GradientBoostingClassifier -> cross validation accuracy: mean
= %0.3f std = %0.3f" % (np.mean(scores), np.std(scores))
Out: GradientBoostingClassifier -> cross validation accuracy: mean =

0.804 std = 0.029
```

In order to obtain the best performance from `GradientBoostingClassifier` and `GradientBoostingRegression`, you have to tweak the following:

- `n_estimators`: Exceeding with estimators, it increases variance. Anyway, if the estimators are not enough, the algorithm will suffer from high bias.
- `max_depth`: It increases the variance and complexity.
- `subsample`: It can effectively reduce variance.
- `learning_rate`: Smaller values can improve optimization in the training process, though it will require more estimators to converge, and thus more computational time.
- `min_samples_leaf`: It can reduce the variance due to noisy data, reserving overfitting to rare cases.

Dealing with big data

Big data challenges data science projects with four points of view: **volume** (data quantity), **velocity**, **variety**, and **veracity**. Scikit-learn package offers a range of classes and functions that will help you effectively work with data so big that it cannot entirely fit in the memory of your computer, no matter what its specifications may be.

Before providing you with an overview of big data solutions, we have to create or import some datasets in order to give you a better idea of the scalability and performances of different algorithms. This will require about 1.5 gigabytes of your hard disk, which will be freed after the experiment.

Creating some big datasets as examples

As a typical example of big data analysis, we will use some textual data from the Internet and we will take advantage of the available `fetch_20newsgroups`, which contains data of 11,314 posts, each one averaging about 206 words, that appeared in 20 different newsgroups:

```
In: import numpy as np
from sklearn.datasets import fetch_20newsgroups
newsgroups_dataset = fetch_20newsgroups(shuffle=True,
remove=('headers', 'footers', 'quotes'), random_state=6)
print 'Posts inside the data: %s' % np.shape(newsgroups_dataset.data)
print 'Average number of words for post: %0.0f' %
np.mean([len(text.split(' ')) for text in newsgroups_dataset.data])
Out: Posts inside the data: 11314
Average number of words for post: 206
```

Instead, to work out a generic classification example, we will create three synthetic datasets that contain from 1,00,000 to up to 10 million cases. You can create and use any of them according to your computer resources. We will always refer to the largest one for our experiments:

```
In: from sklearn.datasets import make_classification
X,y = make_classification(n_samples=10**5, n_features=5,
n_informative=3, random_state=101)
D = np.c_[y,X]
np.savetxt('huge_dataset_10__5.csv', D, delimiter=",") # the saved
file should be around 14,6 MB
del(D, X, y)
X,y = make_classification(n_samples=10**6, n_features=5,
n_informative=3, random_state=101)
D = np.c_[y,X]
np.savetxt('huge_dataset_10__6.csv', D, delimiter=",") # the saved
file should be around 146 MB
del(D, X, y)
```

```
X,y = make_classification(n_samples=10**7, n_features=5,
n_informative=3, random_state=101)
D = np.c_[y,X]
np.savetxt('huge_dataset_10__7.csv', D, delimiter=",") # the saved
file should be around 1,46 GB
del(D, X, y)
```

After creating and using any of the datasets, you can remove them by the following command:

```
import os
os.remove('huge_dataset_10__5.csv')
os.remove('huge_dataset_10__6.csv')
os.remove('huge_dataset_10__7.csv')
```

Scalability with volume

The trick to manage high volumes of data without loading too many megabytes or gigabytes into memory is to incrementally update the parameters of your algorithm until all the observations have been elaborated at least once by the machine learner.

This is possible in Scikit-learn thanks to the `.partial_fit()` method, which has been made available to a certain number of supervised and unsupervised algorithms. Using the `.partial_fit()` method and providing some basic information (for example, for classification, you should know beforehand the number of classes to be predicted), you can immediately start fitting your model, even if you have a single or a few observations.

This method is called `incremental learning`. The chunks of data that you incrementally fed into the learning algorithm are called batches. The critical points of incremental learning are as follows:

- Batch size
- Data preprocessing
- Number of passes with the same examples
- Validation and parameters fine tuning

Batch size generally depends on your available memory. The principle is that the larger the data chunks, the better, since the data sample will get more representatives of the data distributions as its size gets larger.

Also, data preprocessing is challenging. Incremental learning algorithms work well with data in the range of [-1,+1] or [0,+1] (for instance, Multinomial Bayes won't accept negative values). However, to scale into such a precise range, you need to know beforehand the range of each variable. Alternatively, you have to either pass all the data once, and record the minimum and maximum values, or derive them from the first batch, trimming the following observations that exceed the initial maximum and minimum values.

The number of passes can become a problem. In fact, as you pass the same examples multiple times, you help the predictive coefficients converge to an optimum solution. If you pass too many of the same observations, the algorithm will tend to overfit, that is, it will adapt too much to the data repeated too many times. Some algorithms, like the SGD family, are also very sensitive to the order that you propose to the examples to be learned. Therefore, you have to either set their shuffle option (shuffle=True) or shuffle the file rows before the learning starts, keeping in mind that for efficacy, the order of the rows proposed for the learning should be casual.

Validation is not easy either, especially if you have to validate against unseen chunks, validate in a progressive way, or hold out some observations from every chunk. The latter is also the best way to reserve a sample for grid search or some other optimization.

In our example, we entrust the `SGDClassifier` with a log loss (basically a logistic regression) to learn how to predict a binary outcome given 10**7 observations:

```
In: from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import MinMaxScaler
import pandas as pd
import numpy as np
streaming = pd.read_csv('huge_dataset_10__7.csv', header=None,
chunksize=10000)
learner = SGDClassifier(loss='log')
minmax_scaler = MinMaxScaler(feature_range=(0, 1))
cumulative_accuracy = list()
for n,chunk in enumerate(streaming):
    if n == 0:
            minmax_scaler.fit(chunk.ix[:,1:].values)
    X = minmax_scaler.transform(chunk.ix[:,1:].values)
    X[X>1] = 1
    X[X<0] = 0
    y = chunk.ix[:,0]
    if n > 8 :
        cumulative_accuracy.append(learner.score(X,y))
    learner.partial_fit(X,y,classes=np.unique(y))
print 'Progressive validation mean accuracy %0.3f' %
np.mean(cumulative_accuracy)
Out: Progressive validation mean accuracy 0.660
```

First, pandas `read_csv` allows us to iterate over the file by reading batches of 10,000 observations (the number can be increased or decreased according to your computing resources).

We use the `MinMaxScaler` in order to record the range of each variable on the first batch. For the following batches, we will use the rule that if it exceeds one of the limits of [0,+1], they are trimmed to the nearest limit.

Eventually, starting from the 10th batch, we will record the accuracy of the learning algorithm on each newly received batch before using it to update the training. In the end, the accumulated accuracy scores are averaged, offering a global performance estimation.

Keeping up with velocity

There are various algorithms that work with incremental learning.

For classification, we will recall the following:

- sklearn.naive_bayes.MultinomialNB
- sklearn.naive_bayes.BernoulliNB
- sklearn.linear_model.Perceptron
- sklearn.linear_model.SGDClassifier
- sklearn.linear_model.PassiveAggressiveClassifier

For regression, we will recall the following:

- sklearn.linear_model.SGDRegressor
- sklearn.linear_model.PassiveAggressiveRegressor

As for velocity, they are all comparable in speed. You can try for yourself the following script:

```
In: from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.linear_model import Perceptron
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import PassiveAggressiveClassifier
import pandas as pd
import numpy as np
from datetime import datetime
classifiers  = {
'SGDClassifier hinge loss' : SGDClassifier(loss='hinge',
random_state=101),
'SGDClassifier log loss' : SGDClassifier(loss='log',
random_state=101),
'Perceptron' : Perceptron(random_state=101),
'BernoulliNB' : BernoulliNB(),
'PassiveAggressiveClassifier' : PassiveAggressiveClassifier(random_
state=101)
}
huge_dataset = 'huge_dataset_10__6.csv'
for algorithm in classifiers:
    start = datetime.now()
    minmax_scaler = MinMaxScaler(feature_range=(0, 1))
    streaming = pd.read_csv(huge_dataset, header=None, chunksize=100)
    learner = classifiers[algorithm]
    cumulative_accuracy = list()
    for n,chunk in enumerate(streaming):
        y = chunk.ix[:,0]
        X = chunk.ix[:,1:]
        if n > 50 :
            cumulative_accuracy.append(learner.score(X,y))
        learner.partial_fit(X,y,classes=np.unique(y))
    elapsed_time = datetime.now() - start
    print algorithm + ' : mean accuracy %0.3f in %s secs' %
(np.mean(cumulative_accuracy),elapsed_time.total_seconds())
```

```
Out: BernoulliNB : mean accuracy 0.734 in 41.101 secs
Perceptron : mean accuracy 0.616 in 37.479 secs
SGDClassifier hinge loss : mean accuracy 0.712 in 38.43 secs
SGDClassifier log loss : mean accuracy 0.716 in 39.618 secs
PassiveAggressiveClassifier : mean accuracy 0.625 in 40.622 secs
```

> As a general note, remember that smaller batches are slower since that implies more disk access from a database or a file, which is always a bottleneck.

Dealing with variety

Variety is a big data characteristic. This is especially true when we are dealing with textual data or very large categorical variables (for example, variables storing website names in programmatic advertising). As you will learn from batches of examples, and as you unfold categories or words, each one is an appropriate and exclusive variable. You may find it difficult to handle the challenge of variety and the unpredictability of large streams of data. Scikit-learn package provides you with a simple and fast way to implement the hashing trick and completely forget the problem of defining in advance of a rigid variable structure.

The hashing trick uses hash functions and sparse matrices in order to save your time, resources, and hassle. The hash functions are functions that map in a deterministic way any input they receive. It doesn't matter if you feed them with numbers or strings, they will always provide you with an integer number in a certain range. Sparse matrices are, instead, arrays that record only values that are not zero, since their default value is zero for any combination of their row and column. Therefore, the hashing trick bounds every possible input; it doesn't matter if it was previously unseen to a certain range or position on a corresponding input sparse matrix, which is loaded with a value that is not 0.

For instance, if your input is `Python`, a hashing command like `abs(hash('Python'))` can transform that into the 539294296 integer number and then assign the value of 1 to the cell at the 539294296 column index. The hash function is a very fast and convenient way to always express the same column index given the same input. The using of only absolute values assures that each index corresponds only to a column in our array (negative indexes just start from the last column, and hence in Python, each column of an array can be expressed by both a positive and negative number).

The example that follows uses the `HashingVectorizer` class, a convenient class that automatically takes documents, separates the words, and transforms them, thanks to the hashing trick, into an input matrix. The script aims at learning why posts are published in 20 distinct newsgroups on the basis of the words used on the existing posts in the newsgroups:

```
In: import pandas as pd
import numpy as np
from sklearn.linear_model import SGDClassifier
from sklearn.feature_extraction.text import HashingVectorizer
def streaming():
    for response, item in zip(newsgroups_dataset.target,
newsgroups_dataset.data):
        yield response, item
hashing_trick = HashingVectorizer(stop_words='english', norm = 'l2',
non_negative=True)
learner = SGDClassifier(random_state=101)
texts = list()
targets = list()
for n,(target, text) in enumerate(streaming()):
    texts.append(text)
    targets.append(target)
    if n % 1000 == 0 and n >0:
        learning_chunk = hashing_trick.transform(texts)
        if n > 1000:
            last_validation_score = learner.score(learning_chunk,
targets),
        learner.partial_fit(learning_chunk, targets, classes=[k for k
in range(20)])
        texts, targets = list(), list()
print 'Last validation score: %0.3f' % last_validation_score
Out: Last validation score: 0.710
```

At this point, no matter what text you may input, the predictive algorithm will always answer by pointing out a class. In our case, it points out a `newsgroup` suitable for the post to appear on. Let's try out this algorithm with a text taken from a classified ad:

```
In: New_text = ['A 2014 red Toyota Prius v Five with fewer than 14K
miles. Powered by a reliable 1.8L four cylinder hybrid engine that
averages 44mpg in the city and 40mpg on the highway.']
```

```
text_vector = hashing_trick.transform(New_text)
print np.shape(text_vector), type(text_vector)
print 'Predicted newsgroup: %s' % newsgroups_dataset.target_
names[learner.predict(text_vector)]
Out:
(1, 1048576) <class 'scipy.sparse.csr.csr_matrix'>
Predicted newsgroup: rec.autos
```

Naturally, you may change the `New_text` variable and discover where your text will most likely be displayed in a newsgroup. Note that the `HashingVectorizer` class has transformed the text into a `csr_matrix` (which is quite an efficient sparse matrix), having about 1 million columns.

A quick overview of Stochastic Gradient Descent (SGD)

We will close this chapter on big data with a quick overview of the SGD family comprising of SGDClassifier (for classification) and SGDRegressor (for regression).

Like other classifiers, they can be fit by using the `.fit()` method (passing row by row the in-memory dataset to the learning algorithm) or the previously seen `.partial_fit()` method based on batches. In the latter case, if you are classifying, you have to declare the predicted classes with the class parameter. It can accept a list containing all the class code that it should expect to meet during the training phase.

SGDClassifier can behave as a logistic regression when the loss parameter is set to `loss`. It transforms into a linear SVC if the loss is set to `hinge`. It can also take the form of other loss functions or even the loss functions working for regression.

SGDRegressor mimics a linear regression using the `squared_loss` loss parameter. Instead, the huber loss transforms the squared loss into a linear loss over a certain distance epsilon (another parameter to be fixed). It can also act as a linear SVR using the `epsilon_insensitive` loss function or the slightly different `squared_epsilon_insensitive` (which penalizes outliers more).

As in other situations with machine learning, performance of the different loss functions on your data science problem cannot be estimated a priori. Anyway, please take into account that if you are doing classification and you need an estimation of class probabilities, you will be limited in your choice to `log` or `modified_huber` only.

Key parameters that require tuning for this algorithm to best work with your data are:

- n_iter: The number of iterations over the data; the more the passes, the better the optimization of the algorithm. However, there is a higher risk of overfitting. Empirically, SGD tends to converge to a stable solution after having seen 10**6 examples. Given your examples, set your number of iterations accordingly.
- penalty : You have to choose l1, l2, or elasticnet, which are all different regularization strategies, in order to avoid overfitting because of overparametrization (using too many unnecessary parameters leads to the memorization of observations more than the learning of patterns). Briefly, l1 tends to reduce unhelpful coefficients to zero, l2 just attenuates them, and elasticnet is a mix of l1 and l2 strategies.
- alpha: This is a multiplier of the regularization term; the higher the alpha, the more the regularization. We advise you to find the best alpha value by performing a grid search ranging from 10**-7 to 10**-1.
- l1_ratio: The l1 ratio is used for elastic net penalty.
- learning_rate: This sets how much the coefficients are affected by every single example. Usually, it is optimal for classifiers and invscaling for regression. If you want to use invscaling for classification, you'll have to set eta0 and power_t (invscaling = eta0 / (t**power_t)). With invscaling, you can start with a lower learning rate, which is less than the optimal rate, though it will decrease slower.
- epsilon: This should be used if your loss is huber, epsilon_insensitive, or squared_epsilon_insensitive.
- shuffle: If this is True, the algorithm will shuffle the order of the training data in order to improve the generalization of the learning.

A peek into Natural Language Processing (NLP)

This section is not strictly related to machine learning, but it contains some machine learning results in the area of Natural Language Processing. Python has many toolkits to process text data, but the most powerful and complete toolkit is **NLTK**, the Natural Language Tool Kit.

In the following sections, we'll explore its core functionalities. We will work on the English language; for other languages, you will first need to download the language corpora (note that sometimes, languages have no free open source corpora for NLTK).

Word tokenization

Tokenization is the action of splitting the text in words. Chunking the whitespace seems very easy, but it's not, because text contains punctuation and contractions. Let's start with an example:

```
In: my_text = "The coolest job in the next 10 years will be
statisticians. People think I'm joking, but who would've guessed that
computer engineers would've been the coolest job of the 1990s?"

simple_tokens = my_text.split(' ')

print simple_tokens

Out: ['The', 'coolest', 'job', 'in', 'the', 'next', '10', 'years',
'will', 'be', 'statisticians.', 'People', 'think', "I'm", 'joking,',
'but', 'who', "would've", 'guessed', 'that', 'computer', 'engineers',
"would've", 'been', 'the', 'coolest', 'job', 'of', 'the', '1990s?']
```

Here, you can immediately see that something is wrong. The following tokens contain more than a word: `statisticians.` (with the final period), `I'm` (two words), and `would've`, `1990s?` (with the final question mark). Let's see now how NLTK performs better in this task (of course, under the hood, the algorithm is more complex than a simple whitespace chunker):

```
In: import nltk

nltk_tokens = nltk.word_tokenize(my_text)

print nltk_tokens

Out: ['The', 'coolest', 'job', 'in', 'the', 'next', '10', 'years',
'will', 'be', 'statisticians', '.', 'People', 'think', 'I', "'m",
'joking', ',', 'but', 'who', 'would', "'ve", 'guessed', 'that',
'computer', 'engineers', 'would', "'ve", 'been', 'the', 'coolest',
'job', 'of', 'the', '1990s', '?']
```

While executing this, or some other NLTK package calls, in case of an error saying: `"Resource u'tokenizers/punkt/english.pickle' not found."`, just type `nltk.download()` on your console and select to either download everything or browse for the missing resource that triggered the warning.

Here, the quality is better, and each token is associated to a word in the text. Note that `.`, `,` and `?` are tokens, too.

There also exists a sentence tokeniser (see the `nltk.tokenize.punkt` module), but it's rarely used in data science.

Stemming

Stemming is the action of reducing inflectional forms of words and taking the words to their core concepts. For example, the concept behind `is`, `be`, `are`, and `am` is the same. Similarly, the concept behind `go` and `goes`, as well as `table` and `tables`, is the same. The operation of deriving the root concept for each word is named stemming. In NLTK, you can choose the stemmer that you'd like to use (there are several ways to get the root part of words). We'll show you one of them, letting the others in IPython associated with this part of the book:

```
In: from nltk.stem import *

stemmer = LancasterStemmer()

print [stemmer.stem(word) for word in nltk_tokens]

Out: ['the', 'coolest', 'job', 'in', 'the', 'next', '10', 'year',
'wil', 'be', 'stat', '.', 'peopl', 'think', 'i', "'m", 'jok', ',',
'but', 'who', 'would', "'ve", 'guess', 'that', 'comput', 'engin',
'would', "'ve", 'been', 'the', 'coolest', 'job', 'of', 'the', '1990s',
'?']
```

In the example, we used the Lancaster stemmer, which is one of the most powerful and recent algorithms. Checking the result, you will immediately see that it's all lowercased, and `statistician` is associated to its root `stat`. Good job!

Word Tagging

Tagging, or POS-Tagging, is the association between a word (or a token) and its part of a speech tag (POS-Tag). After tagging, you know what (and where) the verbs, adjectives, nouns, and so on, are in the sentence. Even in this case, NLTK makes this complex operation very easy:

```
In: import nltk

print nltk.pos_tag(nltk_tokens)

Out: [('The', 'DT'), ('coolest', 'NN'), ('job', 'NN'), ('in', 'IN'),
('the', 'DT'), ('next', 'JJ'), ('10', 'CD'), ('years', 'NNS'),
('will', 'MD'), ('be', 'VB'), ('statisticians', 'NNS'), ('.', '.'),
('People', 'NNS'), ('think', 'VBP'), ('I', 'PRP'), ("'m", 'VBP'),
('joking', 'VBG'), (',', ','), ('but', 'CC'), ('who', 'WP'),
('would', 'MD'), ("'ve", 'VB'), ('guessed', 'VBN'), ('that', 'IN'),
('computer', 'NN'), ('engineers', 'NNS'), ('would', 'MD'), ("'ve",
'VB'), ('been', 'VBN'), ('the', 'DT'), ('coolest', 'NN'), ('job', 'NN'),
('of', 'IN'), ('the', 'DT'), ('1990s', 'CD'), ('?', '.')]
```

Using the syntax of NLTK, you will get that the `The` token represents a determiner (DT), `coolest` and `job` represent nouns (NN), `in` represents a conjunction, and so on. The association is really detailed; in case of a verb, there are six possible tags, as follows:

- take: `VB` (verb, base form)
- took: `VBD` (verb, past tense)
- taking: `VBG` (verb, gerund)
- taken: `VBN` (verb, past participle)
- take: `VBP` (verb, singular present tense)
- takes: `VBZ` (verb, third person singular present tense)

If you need a more detailed view of the sentence, you may want to use the parse tree tagger to understand its syntactic structure. This operation is rarely used in data science since it's great for sentence-by-sentence analysis.

Named Entity Recognition (NER)

The goal of NER is to recognize tokens associated to people, organizations, and locations. Let's use an example to explain it further:

```
In:
import nltk
text = "Elvis Aaron Presley was an American singer and actor. Born in
Tupelo, Mississippi, when Presley was 13 years old he and his family
relocated to Memphis, Tennessee."
chunks = nltk.ne_chunk(nltk.pos_tag(nltk.word_tokenize(text)))
print chunks
Out:
(S
  (PERSON Elvis/NNP)
  (PERSON Aaron/NNP Presley/NNP)
  was/VBD
  an/DT
  (GPE American/JJ)
  singer/NN
  and/CC
  actor/NN
  ./.
  Born/NNP
  in/IN
  (GPE Tupelo/NNP)
  ,/,
  (GPE Mississippi/NNP)
  ,/,
  when/WRB
  (PERSON Presley/NNP)
  was/VBD
  13/CD
  years/NNS
  old/JJ
  he/PRP
  and/CC
  his/PRP$
  family/NN
  relocated/VBD
  to/TO
  (GPE Memphis/NNP)
  ,/,
  (GPE Tennessee/NNP)
  ./.)
```

An extract of the Wikipedia page on Elvis is analyzed and NER-processed. A few entities that have been recognized by NER are:

- Elvis Aaron Presley – PERSON
- American – GPE (Geo political entity)
- Tupelo, Mississippi – GPE (Geopolitical entity)
- Memphis, Tennessee – GPE (Geopolitical entity)

Stopwords

Stopwords are the least informative pieces (or tokens) in a text since they are the most common words (such as the, it, is, as, and not). Stopwords are often removed and, exactly the way it happens in the feature selection phase, if you remove them, the processing takes less time, less memory, and it is sometimes more accurate.

A list of English stopwords is available in sklearn. For other languages, check NLTK:

```
In: import nltk
from sklearn.feature_extraction import text
stop_words = text.ENGLISH_STOP_WORDS
print stop_words
Out: frozenset(['all', 'six', 'less', 'being', 'indeed', 'over',
'move', 'anyway', 'four', 'not', 'own', 'through', 'yourselves',
'fify', 'where', 'mill', 'only', 'find', 'before', 'one', 'whose',
'system', 'how', ...
```

A complete data science example – text classification

Now, here's a complete example that allows you to categorize each text in the right category. We will use the 20newsgroup dataset, which was already introduced in the first chapter. To make things more realistic and prevent the classifier from overfitting the data, we'll remove email headers, footers (like signature), and quotes. Also in this case, the goal is to classify between two similar categories: sci.med and sci.space. We will use the accuracy measure to evaluate the classification:

```
In: import nltk
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import fetch_20newsgroups
import numpy as np
categories = ['sci.med', 'sci.space']
to_remove = ('headers', 'footers', 'quotes')
twenty_sci_news_train = fetch_20newsgroups(subset='train',
remove=to_remove, categories=categories)
twenty_sci_news_test = fetch_20newsgroups(subset='test',
remove=to_remove, categories=categories)
```

Let's start with the easiest approach to preprocess the textual data—using TfIdf:

```
In: tf_vect = TfidfVectorizer()
X_train = tf_vect.fit_transform(twenty_sci_news_train.data)
X_test = tf_vect.transform(twenty_sci_news_test.data)
y_train = twenty_sci_news_train.target
y_test = twenty_sci_news_test.target
```

Now, let's use a linear classifier (SGDClassifier) to perform the classification task. One last thing to do is to print out the classification accuracy:

```
In: clf = SGDClassifier()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print "Accuracy=", accuracy_score(y_test, y_pred)
Out: Accuracy= 0.878481012658
```

An accuracy of `87.8` percent is a very good result. The entire program consists of less than 20 lines of code. Now, let's see if we can get something better. In this chapter, we've learned about stopword removal, tokenization, and stemming. Let's see if we gain accuracy by using them:

```
In:
def clean_and_stem_text(text):
    tokens = nltk.word_tokenize(text.lower())
    clean_tokens = [word for word in tokens if word not in
stop_words]
    stem_tokens = [stemmer.stem(token) for token in clean_tokens]
    return " ".join(stem_tokens)
cleaned_docs_train = [clean_and_stem_text(text) for text in
twenty_sci_news_train.data]
cleaned_docs_test = [clean_and_stem_text(text) for text in
twenty_sci_news_test.data]
```

The `clean_and_stem_text` function basically lowercases, tokenizes, stems, and reconstructs every document in the dataset. Finally, we will apply the same preprocessing (`Tfidf`) and classifier (`SGDClassifier`) that we used in the preceding example:

```
In: X1_train = tf_vect.fit_transform(cleaned_docs_train)
X1_test = tf_vect.transform(cleaned_docs_test)
clf.fit(X1_train, y_train)
y1_pred = clf.predict(X1_test)
print "Accuracy=", accuracy_score(y_test, y1_pred)
Out: Accuracy= 0.893670886076
```

This processing requires more time, but we gained an accuracy of about 1.5 percent. An accurate tuning of the parameters of the Tfidf and a cross-validated choice of the parameters of the classifier will eventually boost the accuracy to over 90 percent. So far, we're happy with this performance, but you can try to break that barrier.

An overview of unsupervised learning

In all the methods we've seen so far, every sample or observation has its own target label or value. In some other cases, the dataset is unlabelled and, in order to extract the structure of the data, you need an unsupervised approach. In this section, we're going to introduce two methods to perform clustering, as they are among the most used methods for unsupervised learning.

> Keep in mind that often, the terms clustering and unsupervised learning are considered synonymous.

The first method that we'll introduce you to is named K-means. In signal processing, it is the equivalent of a vectorial quantization, that is, the selection of the best codeword (from a given codebook) that better approximates the input observation (or a word).

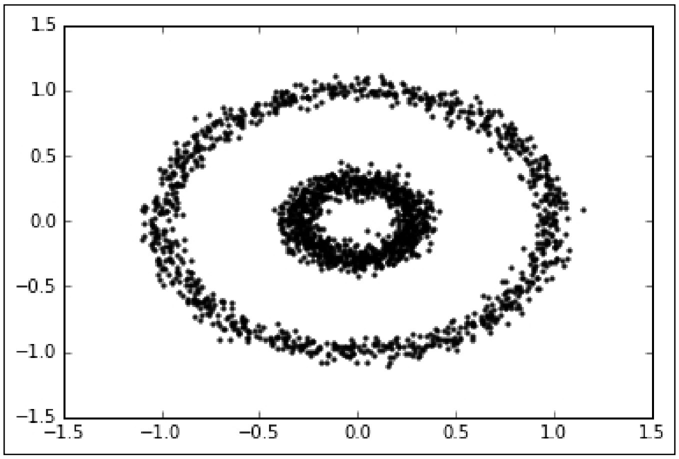
You must provide the algorithm with the K parameter, which is the number of clusters. Sometimes, this might be a limitation because you have to first investigate which is the right K for the current dataset. K-means iterates an EM (expectation/maximization) approach. During the first phase, it assigns each training point to the closest cluster centroid; during the second phase, it moves the cluster centroid to the center of the mass of the points assigned to it (to reduce the distortion). The initial placement of the centroids is random. So, sometimes, you need to run the algorithm several times so as not to find a local minimum.

That's the theory behind the algorithm; now, let's see it in practice. In this section, we're using two 2-dimensional dummy datasets that will explain what's going on better. Both datasets are composed of 2,000 samples so that you can also have an idea about the processing time. The first one contains two (noisy) circles with the same origin but different radii; the second one contains four blobs of points.

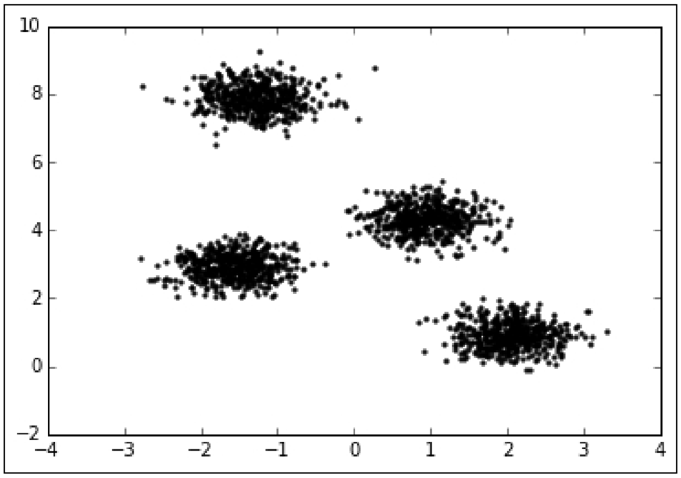
Now, let's create the artificial datasets:

```
In: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
In: from sklearn import datasets
N_samples = 2000
dataset_1 = np.array(datasets.make_circles(n_samples=N_samples,
noise=0.05, factor=0.3)[0])
```

```
dataset_2 = np.array(datasets.make_blobs(n_samples=N_samples, centers=4,
cluster_std=0.4, random_state=0)[0])
In: plt.scatter(dataset_1[:,0], dataset_1[:,1], c='k', alpha=0.8, s=5.0)
plt.show()
```

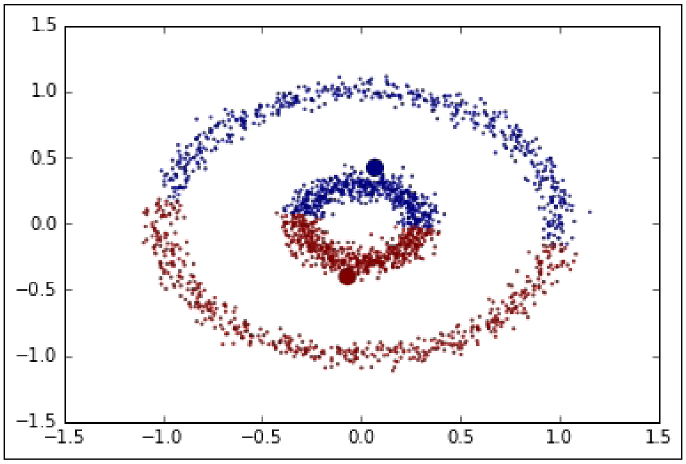


```
In: plt.scatter(dataset_2[:,0], dataset_2[:,1], c='k', alpha=0.8, s=5.0)
plt.show()
```



Now it's the time to apply K-means. We will set `K=2` in this case. Let's see the results:

```
In: from sklearn.cluster import KMeans
K_dataset_1 = 2
km_1 = KMeans(n_clusters=K_dataset_1)
labels_1 = km_1.fit(dataset_1).labels_
In: plt.scatter(dataset_1[:,0], dataset_1[:,1], c=labels_1,
alpha=0.8, s=5.0, lw = 0)
plt.scatter(km_1.cluster_centers_[:,0], km_1.cluster_centers_[:,1],
s=100, c=np.unique(labels_1), lw=0.2)
plt.show()
```

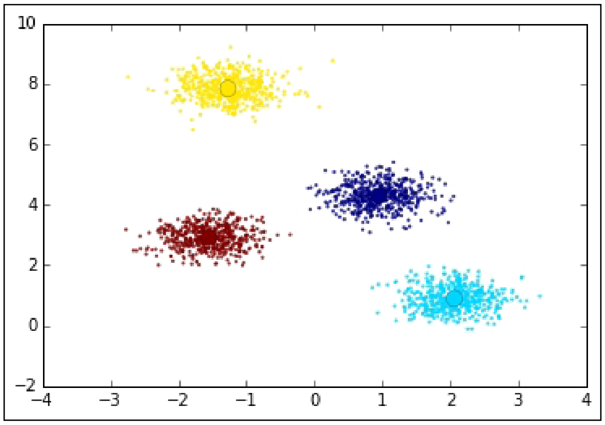


The big dots are the centroids of the two detected clusters. As you can see, K-means is not performing very well on this dataset, because it expects spherical-shaped data clusters. For this dataset, a Kernel-PCA should be applied before using K-means.

Now, let's see how it performs on a spherical-clustered data. In this case, based on our knowledge of the problem and the silhouette coefficient, we will set `K=4`:

```
In: K_dataset_2 = 4
km_2 = KMeans(n_clusters=K_dataset_2)
labels_2 = km_2.fit(dataset_2).labels_
In: plt.scatter(dataset_2[:,0], dataset_2[:,1], c=labels_2,
alpha=0.8, s=5.0, lw = 0)
```

```
plt.scatter(km_2.cluster_centers_[:,0], km_2.cluster_centers_[:,1],
s=100, c=np.unique(labels_2), lw=0.2)
plt.show()
```



In real world cases, you may consider using the Silhouette Coefficient to have an idea about how well-defined the clusters are. You can read more about Silhouette Coefficient at: `http://scikit-learn.org/stable/modules/clustering.html#silhouette-coefficient`.

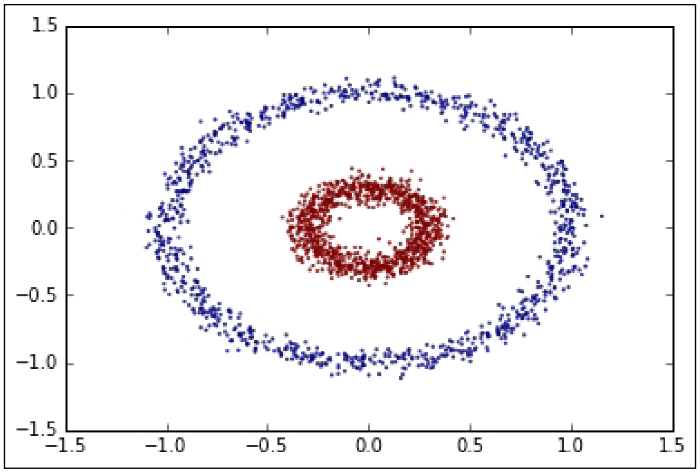
As expected, the result is great. The centroids and clusters are exactly what we had in mind while looking at the unlabelled dataset.

Now, we will introduce you to **DBSCAN**, a density-based clustering technique. It's a very simple technique. It selects a random point; if the point is in a dense area (that is, if it has more than N neighbors), it starts growing the cluster, including all the neighbors and the neighbors of the neighbors, until it reaches a point where there are no more neighbors. If the point is not in a dense area, it is classified as noise. Then, another unlabelled point is randomly selected and the process starts over. This technique is great for nonspherical clusters but it works equally well with spherical ones. The input is just the neighborhood radius (the `eps` parameter, that is, the maximum distance between two points that are being considered neighbors), and the output is the cluster membership label for each point.

Note that the points labeled -1 are classified as noise by DBSCAN.

Let's see an example on the dataset we had previously introduced:

```
In: from sklearn.cluster import DBSCAN
dbs_1 = DBSCAN(eps=0.25)
labels_1 = dbs_1.fit(dataset_1).labels_
In: plt.scatter(dataset_1[:,0], dataset_1[:,1], c=labels_1,
alpha=0.8, s=5.0, lw = 0)
plt.show()
```



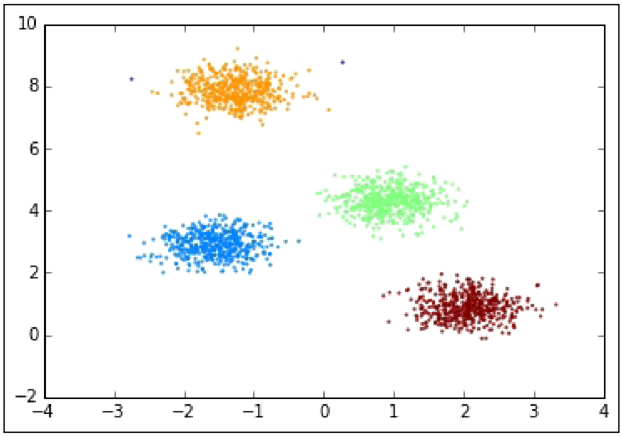
```
In: np.unique(labels_1)
Out: array([0, 1])
```

The result is perfect now. No points have been classified as noise (only the `0` and `1` labels appear in the label set).

Now, let's move on to the other dataset:

```
In: dbs_2 = DBSCAN(eps=0.5)
labels_2 = dbs_2.fit(dataset_2).labels_
```

```
In: plt.scatter(dataset_2[:,0], dataset_2[:,1], c=labels_2,
alpha=0.8, s=5.0, lw = 0)
plt.show()
```



```
In: np.unique(labels_2)
Out: array([-1,  0,  1,  2,  3])
```

It took some time to select the best settings for DBSCAN, and in this case, four clusters have been detected and a few points have been classified as noise (since the label set contains -1).

> At the end of this section, a last important note is that in this essential introduction of K-means and DBSCAN, we have always used the Euclidean distance in examples, as it is the default distance metric in these functions. When using this distance in real cases, remember that you have to normalize each feature (z-normalization) so that every feature contributes equally to the final distortion. If the dataset is not normalized, the features that has larger support will have more decision power on the output label, and that's something that we don't want.

Summary

In this chapter, we introduced the essentials of machine learning. We started with some easy, but still quite effective, classifiers (linear and logistic regressors, Naive Bayes, and K-Nearest Neighbors). Then, we moved on to the more advanced ones (SVM). We explained how to compose weak classifiers together (ensembles, RandomForests, and Gradient Tree Boosting). Finally, we had a peek at the algorithms used in big data and clustering.

In the next chapter, you'll be introduced to **Graphs**, which is an interesting deviation from the predictors/target flat matrices. It is quite a hot topic in data science now. Expect to delve into very complex and intricate networks!

5
Social Network Analysis

Social Network Analysis, usually referred to as **SNA**, models and studies the relationships of a group of social entities, which exist in the form of a network. An entity can be a person, a computer, or a webpage, and a relation can be a like, link, or a friendship (that is, a connection between the entities).

In this chapter, you'll learn about the following:

- Graphs, since social networks are usually represented in this form
- Important algorithms that are used to gain insights from a graph
- How to load, dump, and sample large graphs

Introduction to graph theory

Basically, a graph is a data structure that's able to represent relations in a collection of objects. Under this paradigm, the objects are the graph's nodes and the relations are the graph's links (or edges). The graph is directed if the links have an orientation (conceptually, they're like the one-way streets of a city); otherwise, the graph is undirected. In the following table, examples of well-known graphs are provided:

Graph example	**Type**	**Nodes**	**Edges**
Internet network	Directed	Web pages	Links
Facebook	Undirected	People	Friendship
Twitter	Directed	People	Follower
IP network	Undirected	Hosts	Wires/connections
Navigation systems	Directed	Places/addresses	Streets
Wikipedia	Directed	Pages	Anchor links
Scientific literature	Directed	Papers	Citations
Markov chains	Directed	Status	Emission probability

All the preceding examples can be expressed as relations between nodes like in a traditional RDBMS, such as MySQL or Postgres. Now, to see the advantages of a graph data structure, try to think of how complex the following query in SQL would be for a social network such as Facebook (think about a recommender system that helps you find people you may know):

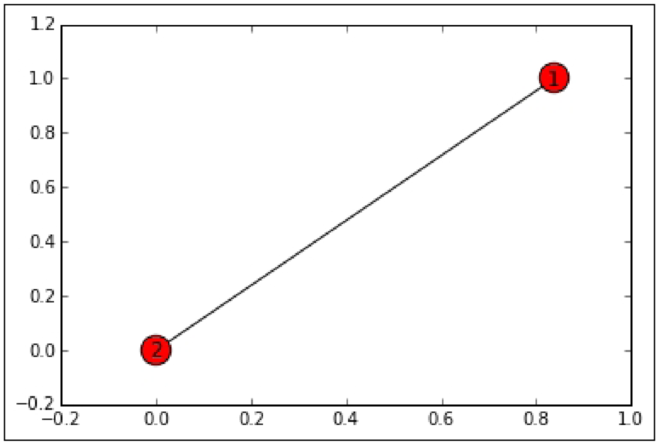
```
Find all people who are friends of my friends, but not my friends
```

Compare the preceding query to the following query on a graph:

```
Get all friends connected to me having distance=2
```

Now, let's see how to create a graph or a social network with Python. The library that we're going to use extensively throughout the chapter is named `NetworkX`. It is capable of handling small to medium graphs and it is complete and powerful.

```
In: %matplotlib inline
import networkx as nx
import matplotlib.pyplot as plt
In: G = nx.Graph()
G.add_edge(1,2)
nx.draw_networkx(G)
plt.show()
```



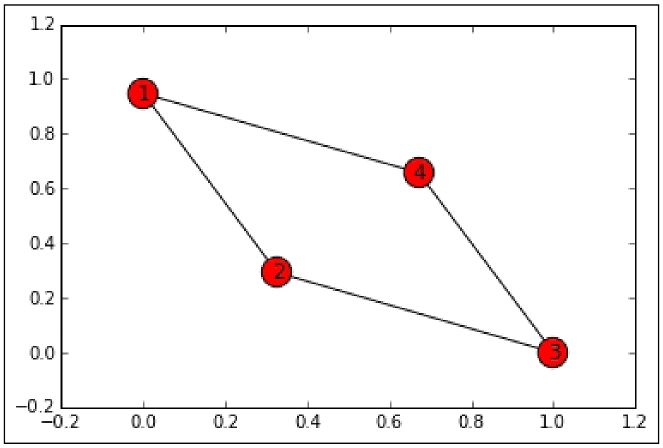
The command is self-explanatory. After the imports, we will first define a (`NetworkX`) graph object (by default, it's an undirected one). Then, we will add an edge (that is, a connection) between two nodes (since the nodes are not already in the graph, they're automatically created). Finally, we will plot the graph. The graph layout (the positions of the nodes) is automatically generated by the library.

With the `.add_note()` method, adding other nodes to the graph is pretty straightforward. For example, if you want to add the nodes `3` and `4`, you can simply use the following code:

```
In: G.add_nodes_from([3, 4])
```

The preceding code will add the two nodes. Since they're not linked to the other nodes, they'll be unconnected. Similarly, to add additional edges to the graph, you can use the following code:

```
In: G.add_edge(3,4)

G.add_edges_from([(2, 3), (4, 1)])

nx.draw_networkx(G)

plt.show()
```



To obtain a collection of nodes in the graph, just use the `.nodes()` method. Similarly, `.edges()` gives you the list of edges as a list of connected nodes:

```
In: G.nodes()

Out: [1, 2, 3, 4]
```

```
In: G.edges()

Out: [(1, 2), (1, 4), (2, 3), (3, 4)]
```

There are several ways to represent (describe) a graph. In the following section, we'll describe the most popular ones. At first, you can use an adjacency list. It lists the neighbors of every node, that is, `list[0]` contains the adjacency nodes of the first node, as in `G.nodes()`:

```
In: G.adjacency_list()
Out: [[2, 4], [1, 3], [2, 4], [1, 3]]
```

To make the description self-contained, you can represent the graph as a dictionary of lists. Here, the nodes' names are the dictionary keys, and their values are the nodes' adjacency lists:

```
In: nx.to_dict_of_lists(G)
Out:
{1: [2, 4], 2: [1, 3], 3: [2, 4], 4: [1, 3]}
```

On the other hand, you can describe a graph as a **collection of edges**. In the output, the third element of each tuple is the attribute of the edge. In fact, every edge can have one or more attributes (such as weight, cardinality, and so on). Since we created a very simple graph, in following example, we have no attributes.

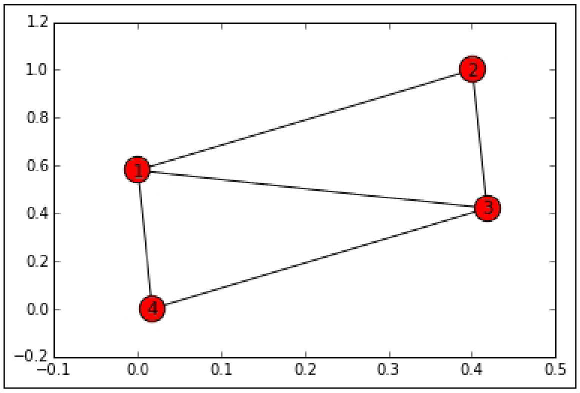
```
In: nx.to_edgelist(G)
Out: [(1, 2, {}), (1, 4, {}), (2, 3, {}), (3, 4, {})]
```

Finally, a graph can be described as a NumPy matrix. If the matrix contains a 1 in the `(i, j)` position, it means that there is a link between the `i` and `j` nodes. Since the matrix usually contains very few ones (compared to the number of zeros), it's usually represented as a sparse (SciPy) matrix. Please note that the matrix description is exhaustive. Therefore, undirected graphs are transformed to directed ones, and a link connecting `(i, j)` is transformed to two links, `(i, j)` and `(j, i)`. Thus, a symmetric matrix is created, as in the following example:

```
In: nx.to_numpy_matrix(G)
Out:
matrix([[ 0.,  1.,  0.,  1.],
        [ 1.,  0.,  1.,  0.],
        [ 0.,  1.,  0.,  1.],
        [ 1.,  0.,  1.,  0.]])


In: print nx.to_scipy_sparse_matrix(G)
Out:
  (0, 1)  1
  (0, 3)  1
  (1, 0)  1
  (1, 2)  1
  (2, 1)  1
  (2, 3)   1
  (3, 0)   1
  (3, 2)   1
```

An important measure of a node in a graph is its degree. In an undirected graph, it represents the number of links the node has. For directed graphs, there are two types of degree: in-degree and out-degree. These respectively count the inbound and outbound links of the node. Let's add a node (to unbalance the graph) and calculate the nodes' degrees, as follows:

```
In: G.add_edge(1,3)
nx.draw_networkx(G)
plt.show()
```



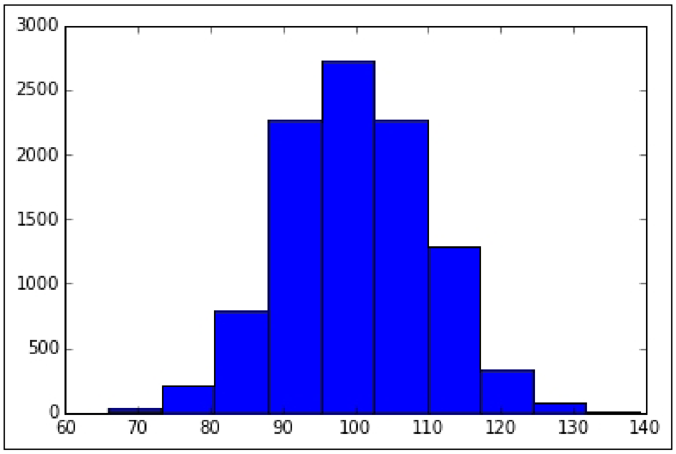
Figures in this chapter can be different to the ones obtained on your local computer because graphical layout initialization is made with random parameters.

The degree of the nodes is displayed as follows:

```
In: G.degree()
Out: {1: 3, 2: 2, 3: 3, 4: 2}
```

For large graphs, this measure is impractical since the output dictionary has an item for every node. In such cases, a histogram of the nodes' degree is often used to approximate its distribution. In the following example, a random network with 10,000 nodes and a link probability of 1 percent is built. Then, the histogram of the node degree is extracted and shown, as follows:

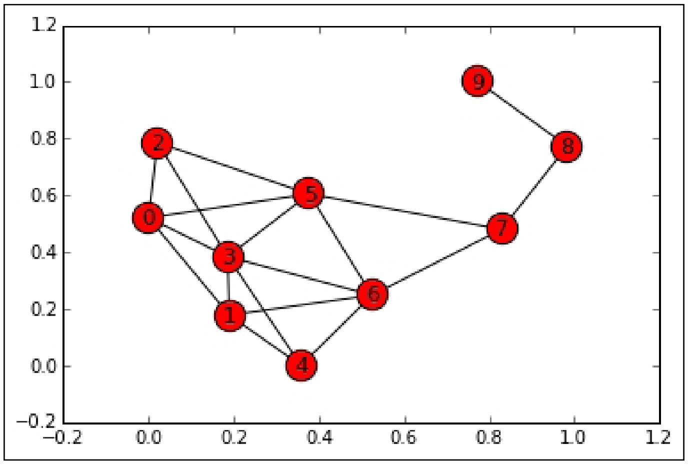
```
In: plt.hist(nx.fast_gnp_random_graph(10000, 0.01).degree().values())
```



Graph algorithms

To get insights from graphs, many algorithms have been developed. In this chapter, we'll use a well-known graph in `NetworkX`, the `Krackhardt Kite` graph. It is a dummy graph containing 10 nodes, and it is typically used to proof graph algorithms. Krackhardt is the creator of the structure, which has the shape of a kite. It's composed of two different zones. In the first zone (composed of nodes 0 to 6), the nodes are interlinked; in the other zone (nodes 7 to 9), they are connected as a chain:

```
In: G = nx.krackhardt_kite_graph()
nx.draw_networkx(G)
plt.show()
```



Let's start with connectivity. Two nodes of a graph are connected if there is at least a path (that is, a sequence of traversed nodes) between them.

Note that in a directed graph, you must follow the link's directions.

If at least a path exists, the shortest path between the two nodes is the one with the shortest collection of traversed nodes.

In `NetworkX`, checking whether a path exists between two nodes, calculating the shortest route, and getting its length is very easy. For example, to check the connectivity and the path between the nodes `1` and `9`, you can use the following code:

```
In: print nx.has_path(G, source=1, target=9)
print nx.shortest_path(G, source=1, target=9)
print nx.shortest_path_length(G, source=1, target=9)
Out:
True
[1, 6, 7, 8, 9]
4
```

We will now talk about node centrality. It roughly represents the importance of the node inside the network. It gives an idea about how well the node connects the network. There are multiple types of centrality. We will discuss the betweenness centrality, degree centrality, closeness centrality, and the eigenvector centrality.

- **Betweenness centrality**: This type of centrality gives you an idea about the number of shortest paths in which the node is present. Nodes with high betweenness centrality are the core components of the network, and many shortest paths route through them. In the following example, `NetworkX` offers a straightforward way to compute the betweenness centrality of all the nodes, as follows:

  ```
  In: nx.betweenness_centrality(G)
  Out:
  {0: 0.023148148148148143,
   1: 0.023148148148148143,
   2: 0.0,
   3: 0.10185185185185183,
   4: 0.0,
   5: 0.23148148148148148,
   6: 0.23148148148148148,
   7: 0.38888888888888884,
   8: 0.2222222222222222,
   9: 0.0}
  ```

  As you can imagine, the highest betweenness centrality is achieved by the node `7`. It seems very important since it's the only node that connects the elements `8` and `9` (it's their gateway to the network). On the contrary, nodes like `9`, `2`, and `4` are on the extreme border of the network, and they are not present in any of the shortest paths of the network. Therefore, these nodes can be removed without affecting the connectivity of the network.

- **Degree centrality**: This type of centrality is simply the percentage of the vertexes that are incident upon a node. Note that in directed graphs, there are two degree centralities for every node: in-degree and out-degree centrality. Let's have a look at the following example:

  ```
  In: nx.degree_centrality(G)
  Out:
  {0: 0.4444444444444444,
   1: 0.4444444444444444,
   2: 0.3333333333333333,
  ```

```
 3: 0.6666666666666666,
 4: 0.3333333333333333,
 5: 0.5555555555555556,
 6: 0.5555555555555556,
 7: 0.3333333333333333,
 8: 0.2222222222222222,
 9: 0.1111111111111111}
```

As expected, node `3` has the highest degree centrality since it's the node with the maximum number of links (it's connected to six other nodes). On the contrary, node `9` is the node with the lowest degree since it has only one edge.

- **Closeness centrality**: To compute this for every node, calculate the shortest path distance to all other nodes, average it, divide the average by the maximum distance, and take the inverse of that value. It results in a score between 0 (the greater average distance) and 1 (the lower average distance). In our example, for node `9`, the shortest path distances are [1, 2, 3, 3, 4, 4, 4, 5, 5]. The average (3.44) is then divided by 5 (the maximum distance) and subtracted from 1, resulting in a closeness centrality score of 0.31. You can use the following code to compute the closeness centrality for all the nodes in the example graph:

```
In: nx.closeness_centrality(G)
Out:
{0: 0.5294117647058824,
 1: 0.5294117647058824,
 2: 0.5,
 3: 0.6,
 4: 0.5,
 5: 0.6428571428571429,
 6: 0.6428571428571429,
 7: 0.6,
 8: 0.42857142857142855,
 9: 0.3103448275862069}
```

The nodes with high closeness centrality are `5`, `6`, and `3`. In fact, they are the nodes that are present in the middle of the network and, on an average, they can reach all the other nodes with a few hops. The lowest score belongs to node `9`. In fact, its average distance to reach all the other nodes is pretty high.

- **Eigenvector centrality**: If the graph is directed, the nodes represent web pages and the edges represent page links. A slightly modified version is named pagerank. This metric, invented by Larry Page, is the core ranking algorithm of Google. It gives to every node a measure of how important the node is from the point of view of a random surfer. Its name derives from the fact that if you think of the graph as a Markov Chain, the graph represents the eigenvector associated with the greatest eigenvalue. Therefore, from this point of view, this probabilistic measure represents the static distribution of the probability of visiting a node. Let's have a look at the following example:

  ```
  In: nx.eigenvector_centrality(G)
  Out:
  {0: 0.35220918419838565,
   1: 0.35220918419838565,
   2: 0.28583482369644964,
   3: 0.481020669200118,
   4: 0.28583482369644964,
   5: 0.3976909028137205,
   6: 0.3976909028137205,
   7: 0.19586101425312444,
   8: 0.04807425308073236,
   9: 0.011163556091491361}
  ```

  In this example, nodes `3` and `9` respectively have the highest and the lowest scores according to the eigenvector centrality measure. Compared to the degree centrality, the eigenvalue centrality gives an idea about the static distribution of the surfers across the network because it considers, for each node, not only the directly connected neighbors (as in the degree centrality), but also the whole structure of the network. If the graph represented web pages and their connections, this makes them the most/least (probable) visited pages.

As a concluding topic, we'll introduce you to the clustering coefficient. In brief, it is the proportion of the node's neighbors that are also neighbors with each other (that is, the proportion of possible triplets or triangles that exists). Higher values indicate higher cliquishness. It's named this way because it represents the degree to which nodes tends to cluster together. Let's have a look at the following example:

```
In: nx.clustering(G)
Out:
{0: 0.6666666666666666,
 1: 0.6666666666666666,
 2: 1.0,
 3: 0.5333333333333333,
 4: 1.0,
 5: 0.5,
 6: 0.5,
 7: 0.3333333333333333,
 8: 0.0,
 9: 0.0}
```

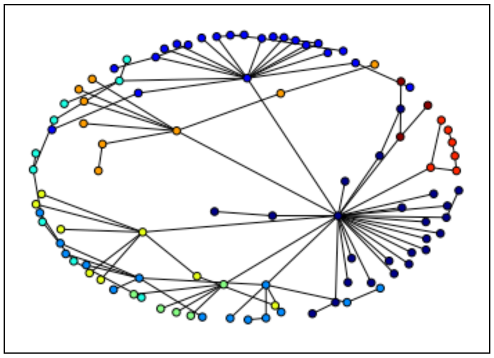
Higher values are seen in the highly connected sections of the graph and lower values in the least connected areas.

Now, let's look at the way by which you can partition the network into multiple subnetworks of nodes. One of the most used algorithms is the Louvain method, which was specifically created to accurately detect communities in large graphs (with a million nodes). We will first introduce you to the modularity measure. It's a measure of the structure of the graph (it's not node-oriented), whose formal math definition is very long and complex and which is beyond the scope of this book (readers can find more information at `https://sites.google.com/site/findcommunities/`). It intuitively measures the quality of the division of a graph into communities, comparing the actual community linkage with a random one. Modularity score falls between -0.5 and +1.0; the higher the value, the better is the division (there is a dense intragroup connectivity, and a sparse intergroup connectivity).

It's a two-step iterative algorithm, first a local optimization, then a global one, then the local again, and so on. In the first step, the algorithm locally maximizes the modularity of small communities. Then, it aggregates the nodes of the same community and hierarchically builds a graph whose nodes are the communities. The method repeats these two steps iteratively until the maximum global modularity score is reached.

To take a peek at this algorithm in a practical example, we need to first create a larger graph. Let's consider a random network with 100 nodes. In this example, we will build the graph with the `powerlaw` algorithm, which tries to maintain an approximate average clustering. For every new node added to the graph, an `m` number of random edges are also added to it and each of them has a probability of `p` to create a triangle. The source code is not included in `NetworkX`, but it's in a separate module named `community.py`. An implementation of this algorithm is shown in the following example:

```
In:
import community


# Creating a powerlaw graph with 100 node, m=1 and p=0.4
G = nx.powerlaw_cluster_graph(100, 1, 0.4)
partition = community.best_partition(G)


for i in set(partition.values()):
   print "Community", i
   members = list_nodes = [nodes for nodes in partition.keys() if
partition[nodes] == i]
   print members


values = [partition.get(node) for node in G.nodes()]
nx.draw_spring(G, cmap = plt.get_cmap('jet'), node_color = values, node_
size=30, with_labels=False)
plt.show()
Out:
print "Modularity score:", community.modularity(partition, G)
Community 0
[0, 1, 8, 12, 17, 20, 22, 24, 26, 35, 41, 47, 49, 50, 56, 65, 70, 75, 79,
82, 87, 88, 90, 91]
Community 1
[2, 6, 7, 9, 11, 18, 23, 29, 34, 36, 37, 40, 48, 55, 60, 71, 85, 89, 94,
95, 96, 98]
[…]
```



```
Modularity score: 0.753290480563
```

The first output of the program is the list of the communities detected in the graph (each community is a collection of nodes). In this case, the algorithm detected eight groups. We wanted to highlight that we didn't specify the number of output communities that we were looking for, but it was automatically decided by the algorithm.

Then, we printed the graph, assigning a different color to each community. You can see that the colors are pretty homogeneous on the edge nodes.

Lastly, the algorithm returns the modularity score of the solution: 0.75 (that's a pretty high score).

Graph loading, dumping, and sampling

Beyond `NetworkX`, graphs and networks can be generated and analyzed with other software. One of the best open source multi-platform software that can be used for their analysis is named Gephi. It's a visual tool and it doesn't require programming skills. It's freely available at `http://gephi.github.io`.

As in machine learning datasets, even graphs have standard formats for their storing, loading, and exchanging. In this way, you can create a graph with `NetworkX`, dump it to a file, and then load and analyze it with Gephi.

One of the most frequently used formats is **Graph Modeling Language** (**GML**). Now, let's see how to dump a graph to GML file:

```
In:
dump_file_base = "dumped_graph"
```

```
# Be sure the dump_file file doesn't exist
def remove_file(filename):
    import os
    if os.path.exists(filename):
        os.remove(filename)

In: G = nx.krackhardt_kite_graph()

In:
# GML format write and read
GML_file = dump_file_base + '.gml'
remove_file(GML_file)

nx.write_gml(G, GML_file)
G2 = nx.read_gml(GML_file)

assert(G.edges() == G2.edges())
```

In the preceding chunk of code, we first removed the dumped file, if it did exist in the first place. Then, we created a graph (the Kite), and after that, we dumped and loaded it. Finally, we compared the original and the loaded structure, asserting that they're equal.

Beyond GML, there are a variety of formats. Each of the formats has different features. Note that some of them remove information pertaining to the network (like edge / node attributes). Similar to the `write_gml` function and its equivalent `read_gml` are the following ones (the names are self-explanatory):

- The adjacency list (`read_adjlist` and `write_adjlist`)
- The multiline adjacency list (`read_multiline_adjlist` and `write_multiline_adjlist`)
- The edge list (`read_edgelist` and `write_edgelist`)
- GEXF (`read_gexf` and `write_gexf`)
- Pickle (`read_gpickle` and `write_gpickle`)
- GraphML (`read_graphml` and `write_graphml`)
- LEDA (`read_leda` and `parse_leda`)
- YAML (`read_yaml` and `write_yaml`)
- Pajek (`read_pajek` and `write_pajek`)

- GIS Shapefile (read_shp and write_shp)
- JSON (load/loads and dump/dumps provides JSON serialization)

The last topic of this chapter is sampling. Why sample a graph? We sample graphs because working with large graphs is sometimes impractical (remember that in the best case, the processing time is proportional to the graph size). Therefore, it's better to sample it, create an algorithm by working on the small-scale scenario, and then test it on the full-scale problem. There are several ways to sample a graph. Here, we're going to introduce you to the three most frequently used techniques.

In the first technique, which is known as node sampling, a limited subset of nodes, along with their links, form the sampled set. In the second technique, which is known as link sampling, a subset of links forms the sampled set. Both these methods are simple and fast, but they might potentially create a different structure of the network. The third method is named snowball sampling. The initial node, all its neighbors, and the neighbors of the neighbors (expanding the selection this way until we reach the *maximum traversal depth* parameter) form the sampled set. In other words, the selection is like a rolling snowball.

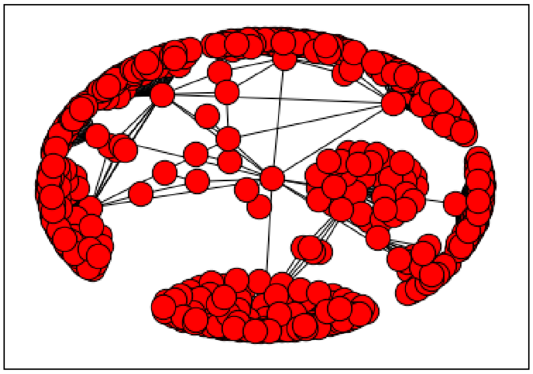
> Note that you can also subsample the traversed links. In other words, each link has a probability of p that has to be followed and selected in the output set.

The last sampling method is not a part of NetworkX, but you can find an implementation for the same in the snowball_sampling.py file.

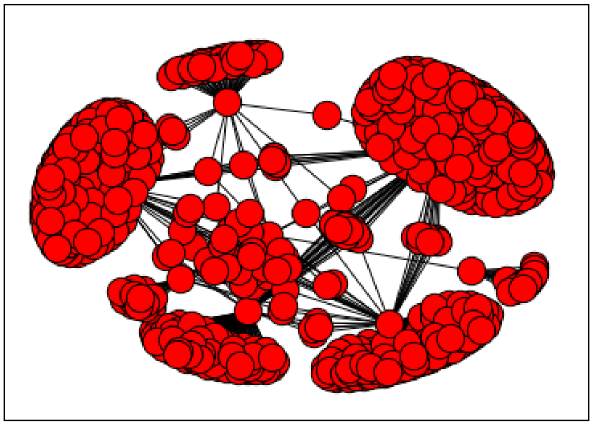
In this example, we will subsample the LiveJournal network by starting with the person with an alberto ID and then expanding recursively twice (in the first example) and three times (in the second example). In the latter instance, every link is followed by a probability of 20 percent, thus decreasing the retrieval time. Here is an example that demonstrates the same:

```
In:
import snowball_sampling
my_social_network = nx.Graph()
snowball_sampling.snowball_sampling(my_social_network, 2, 'alberto')
nx.draw(my_social_network)
plt.show()
Out:
Reaching depth 0
 new nodes to investigate: ['alberto']
```

```
Reaching depth 1
 new nodes to investigate: ['its_kerrie_duhh', 'ph8th',
  'nightraven', 'melisssa', 'mischa', 'deifiedsoul', 'cookita',
    '______eric_', 'seraph76', 'msliebling', 'hermes3x3',
      'eldebate', 'adriannevandal', 'clymore']
```



```
In: my_sampled_social_network = nx.Graph()
snowball_sampling.snowball_sampling(my_sampled_social_network, 3,
'alberto', sampling_rate=0.2)
nx.draw(my_sampled_social_network)
plt.show()
Out:
Reaching depth 0
 new nodes to investigate: ['alberto']
Reaching depth 1
 new nodes to investigate: ['its_kerrie_duhh', 'ph8th', 'nightraven',
  'melisssa', 'mischa', 'deifiedsoul', 'cookita', '______eric_',
    'seraph76', 'msliebling', 'hermes3x3', 'eldebate',
      'adriannevandal', 'clymore']
Reaching depth 2
 new nodes to investigate: ['torcboy', 'flower899', 'inbredhatred',
  'cubnurse', 'motleyprose', 'djbloodrose', 'skullosvibe',
    'necro_man', 'jagbear', 'eeyoredung', 'bearsbearsbears',
      'djmrswhite', 'moonboynm', 'vianegativa', 'blktalon',
        'chironae', 'grimmdolly', 'morpheusnaptime',
          'handelwithcare', 'robdeluxe', 'popebuck1', 'leafshimmer',
            'herbe', 'jeffla', 'rhyno1975', 'needleboy', 'penaranda',
              'maigremeg', 'stargirlms', 'paladincub21', 'rawsound',
                'moroccomole', 'heidilikesyou', 'arshermetica',
                  'leashdog', 'apollonmk', 'greekcub', 'drubear',
                    'gregorbehr', 'trickytoro']
```



Summary

In this chapter, we learned what a social network is—its creation and modification, representation, and some of the important measures of the social network and its nodes. Finally, we discussed the loading and saving of large graphs and ways to deal with the same.

With this chapter, all the essential data science algorithms have been presented; machine learning techniques were discussed in the previous chapter and social network analysis methods in the current one.

In the next chapter, which is the concluding one, we are going to introduce you to the basics of visualization with Matplotlib.

6
Visualization

Last but not least, we are going to illustrate how to create visualizations with Python to support your data science project. Visualization plays an important role in helping you communicate the results and insights derived from data and the learning process.

In this chapter, you will learn how to:

- Use the basic pyplot functions from the matplotlib package
- Leverage pandas DataFrame for **Explorative Data Analysis** (**EDA**)
- Visualize the machine learning and optimization processes we discussed in *Chapter 3*, *The Data Science Pipeline*, and *Chapter 4*, *Machine Learning*
- Understand and visually communicate variables' importance and their relationship with the target outcome

Introducing the basics of matplotlib

Visualization is a fundamental aspect of data science, allowing data scientists to better and more effectively communicate their findings to the organization they operate in, to both data experts and nonexperts. Providing the nuts and bolts of the principles behind communicating information and crafting engaging beautiful visualizations is beyond the scope of our book, but we can recommend resources. For basic visualization rules, you can visit `http://lifehacker.com/5909501/how-to-choose-the-best-chart-for-your-data`. We also recommend the books of Prof. Edward Tufte on analytic design and visualization.

We can instead provide a fast and to-the-point series of essential recipes that can get you started with visualization using Python and which you can refer to anytime you need to create a specific graphic chart.

The matplotlib is a Python package for plotting graphics. Created by John Hunter, it has been developed in order to address a lack of integration between Python and external software with graphical capabilities, such as MATLAB or gnuplot. Greatly influenced by MATLAB, matplotlib presents a similar syntax. In particular, the `matplotlib.pyplot` module will be the core of our essential introduction to a few indispensable graphical tools to represent your data and analysis.

Each pyplot command makes a change on a figure. Once you set a figure, all additional commands will operate on it. Thus, it is easy to incrementally improve and enrich our graphic representation. All the examples we present are therefore expressed in commented building blocks so that you can later draft your basic representation and then look for specific commands in our examples in order to improve it as you plan.

With the `pyplot.figure()` command, you can initialize a new visualization, though it suffices to call a plotting command to automatically start it. Instead, by using `pyplot.show()`, you close the figure that you were operating on and you can open new figures.

Before starting with a few visualization examples, let's import the necessary packages:

```
In: import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
```

In this way, we can always refer to pyplot, the MATLAB-like module, as `plt` and access the complete matplotlib functionality set with the help of `mpl`.

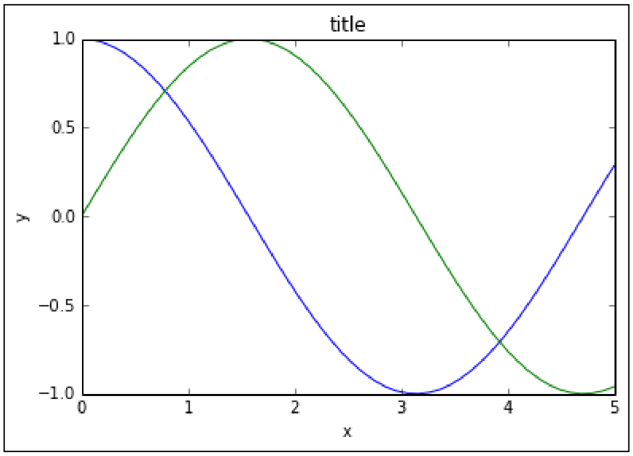
Curve plotting

Our first problem will require you to draw a function with pyplot. Drawing a function is very simple; you just have to get a series of *x* coordinates and map them to the *y* axis by using the function that you want to plot. Since the mapping results are stored away in two vectors, the `plot` function will deal with the curve representation whose precision will be greater if the mapped points are enough (50 points is a good sampling number):

```
In: import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 5, 50)
y_cos = np.cos(x)
y_sin = np.sin(x)
```

Using the NumPy `linspace()` function, we will create a series of 50 equally distanced numbers ranging from zero to five. We can use them to map our *y* to the cosine and sine functions:

```
In: plt.figure() # initialize a figure
plt.plot(x,y_cos) # plot series of coordinates as a line
plt.plot(x,y_sin)
plt.xlabel('x') # adds label to x axis
plt.ylabel('y') # adds label to y axis
plt.title('title') # adds a title
plt.show() # close a figure
```



The `pyplot.plot` command can plot more curves in a sequence, with each curve taking a different color according to an internal color schema, which can be customized by explicating the favored color sequence:

```
In: rcParams['axes.color_cycle'] = ['red', 'blue', 'green']
```

Moreover the `plot` command, if not given any other information, will assume that you are going to plot a line. So, it will link all the provided points into a curve. If you add a new parameter like `'.'`, that is, `plt.plot(x,y_cos,'.')`, you signal that you instead want to plot a series of separated points (the string for a line is `'-'`, but we will soon show another example).

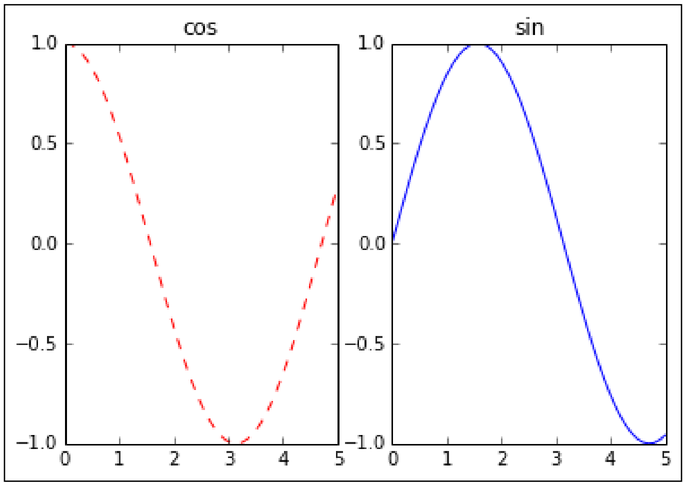
In this way, the next graphs will first have a red curve, the second will be blue, and the third green. Then, the color loop will restart.

Please note that you can also set the title of the graph and label the axis by the title, `xlabel`, and `ylabel` from pyplot.

Using panels

This second exemplification will show you how to create more graphic panels and plot on each of them. We will also try to personalize your curves more by using different colors, sizes, and styles. Here is the example:

```
In: import matplotlib.pyplot as plt

In: plt.subplot(1,2,1) # defines 1 row 2 column panel, activates
figure 1

plt.plot(x,y_cos,'r--')

plt.title('cos') # adds a title

plt.subplot(1,2,2) # defines 1 row 2 column panel, activates figure 2

plt.plot(x,y_sin,'b-')

plt.title('sin')

plt.show()
```



The subplot command accepts the `subplot(nrows, ncols, plot_number)` parameter form. Therefore, it reserves a certain amount of space for the representation based on the `nrows` and `ncols` parameters and plots on the `plot_number` area (starting from the area `1` on the left).

You can also accompany the `plot` command coordinates with another string parameter, which is useful for the defining of color and the type of the curve. The strings work by combining the codes that you can find on the following links:

- `http://matplotlib.org/api/lines_api.html#matplotlib.lines.Line2D.set_linestyle` will present the different line styles.

- `http://matplotlib.org/api/colors_api.html` offers a complete overview of the basic built-in colors. The page also points out that you can either use the `color` parameter together with the html names or hex strings for colors, or define the color you desire by using an RGB tuple, where each value of the tuple lies in the range of [0,1]. For instance, a valid parameter is `color = (0.1,0.9,0.9)`.
- `http://matplotlib.org/api/markers_api.html` lists all the possible marker styles you can adopt for your points.

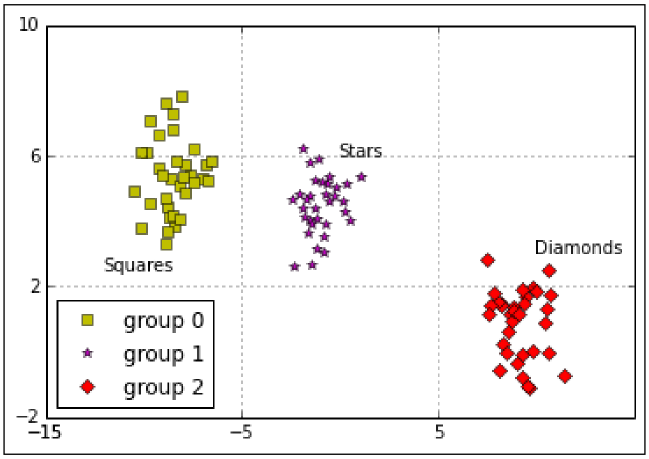
Scatterplots

Scatterplots plot two variables as points on a plane, and they can help you figure out the relationship between the two variables. They are also quite effective if you want to represent groups and clusters. In our example, we will create three data clusters and represent them in a scatterplot with different shapes and colors:

```
In: from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt


D = make_blobs(n_samples=100, n_features=2, centers=3,
random_state=7)
groups = D[1]
coordinates = D[0]
```

Since we have to plot three different groups, we will have to use three distinct `plot` commands. Each command specifies a different color and shape (the `'ys'`, `'m*'`, `'rD'` strings, where the first letter is the color and the second is the marker). Please also note that each plot instance is marked by a `label` parameter, which is used to assign a name to the group that has to be reported later in a legend:

```
In: plt.plot(coordinates[groups==0,0], coordinates[groups==0,1],
'ys', label='group 0') # yellow square
plt.plot(coordinates[groups==1,0], coordinates[groups==1,1], 'm*',
label='group 1') # magenta stars
plt.plot(coordinates[groups==2,0], coordinates[groups==2,1], 'rD',
label='group 2') # red diamonds
plt.ylim(-2,10) # redefines the limits of y axis
plt.yticks([10,6,2,-2]) # redefines y axis ticks
plt.xticks([-15,-5,5,-15]) # redefines x axis ticks
plt.grid() # adds a grid
plt.annotate('Squares', (-12,2.5)) # prints text at coordinates
plt.annotate('Stars', (0,6))
plt.annotate('Diamonds', (10,3))
plt.legend(loc='lower left', numpoints= 1) # places a legend of
labelled items
plt.show()
```

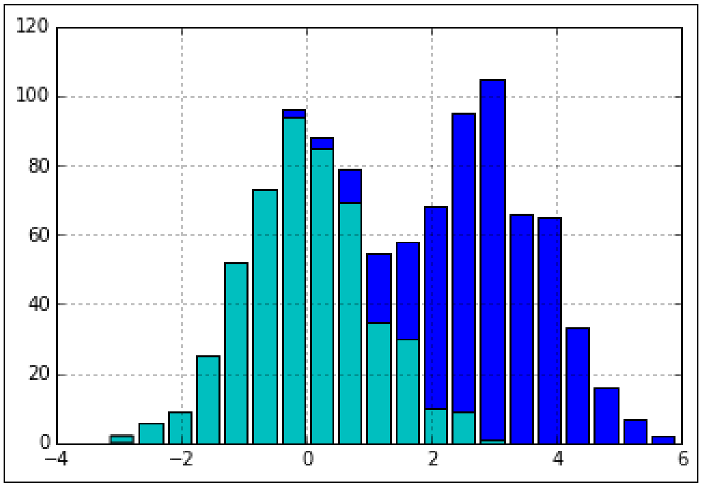


Our scatterplot is ready. We have also added a legend (`pyplot.legend`), fixed a limit for both the axes (`pyplot.xlim` and `pyplot ylim`), and precisely explicated the ticks (`plt.xticks` and `plt.yticks`) that had to be put on them by specifying a list of values. Therefore, the grid (`pyplot.grid`) divides the plot exactly into nine quadrants and allows you to have a better idea of where the groups are positioned. We then printed some text with the group names (`pyplot.annotate`).

Histograms

Histograms can effectively represent the distribution of a variable. Here, we will visualize two normal distributions—both with unit standard deviation, but one with a mean of zero and the other with a mean of 3.0:

```
In: import numpy as np
import matplotlib.pyplot as plt
x = np.random.normal(loc=0.0, scale=1.0, size=500)
z = np.random.normal(loc=3.0, scale=1.0, size=500)
plt.hist(np.column_stack((x,z)), bins=20, histtype='bar', color =
['c','b'], stacked=True)
plt.grid()
plt.show()
```



There are a few ways to personalize this kind of plot and obtain more insight about the analyzed distributions.

First of all, by changing the number of bins, you will change how the distributions are discretized. Generally, 10 to 20 bins offer a good understanding of the distribution, though it really depends on the size of the dataset as well as the distribution. For instance, the Freedman–Diaconis rule prescribes that the optimal number of bins in a histogram depends on the bin's width, which can be calculated using the interquartile range (IQR) and the number of observations:

$$h = 2 * IQR * n^{-1/3}$$

Having calculated *h*, which is the bin-width, the number of bins can therefore be computed by dividing the difference between the maximum and the minimum value by *h*:

$$bins = \left(max - min \right) / h$$

We can also change the type of visualization from bars to steps by changing the parameters from *histtype*='*bar*' to *histtype*='*step*'. By changing the Boolean parameter `stacked` to False, the curves won't stack into a unique bar in the parts that overlap, but you will clearly see the separate bars of each one.

Bar graphs

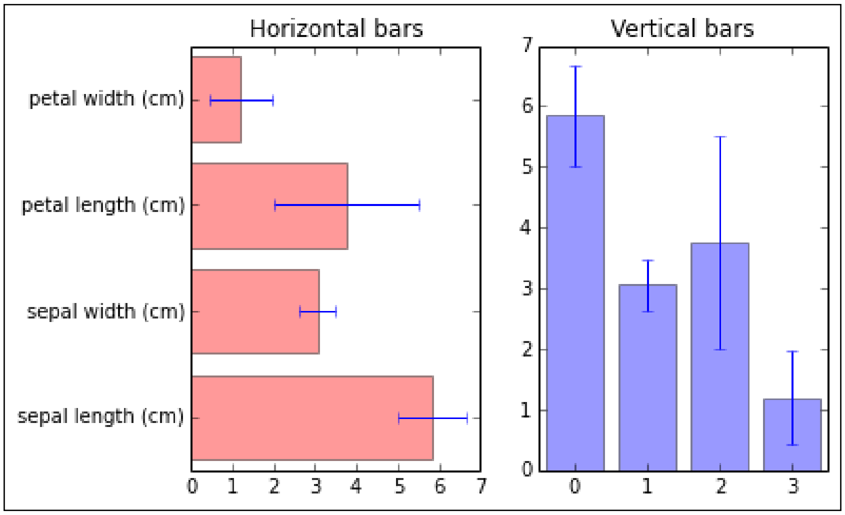
Bar graphs are quite useful. They can be arranged either horizontally or vertically to present mean estimate and error bands. They can be used to present various statistics of your predictors and how they relate to the target variable.

In our example, we will present the mean and standard deviation for the four variables of the Iris dataset:

```
In: from sklearn.datasets import load_iris
import numpy as np
import matplotlib.pyplot as plt
iris = load_iris()
average = np.mean(iris.data, axis=0)
std     = np.std(iris.data, axis=0)
range_  = range(np.shape(iris.data)[1])
```

In our representation, we will prepare two subplots—one with horizontal bars (`plt.barh`), and the other with vertical bars (`plt.bar`). The standard error is represented by an error bar, and according to the graph orientation, we can use the `xerr` parameter for horizontal bars and `yerr` for the vertical ones:

```
In: subplot(1,2,1) # defines 1 row 2 column panel, activates figure
1
plt.title('Horizontal bars')
plt.barh(range_,average, color="r", xerr=std, alpha=0.4,
align="center")
plt.yticks(range_, iris.feature_names)
subplot(1,2,2) # defines 1 row 2 column panel, activates figure 2
plt.title('Vertical bars')
plt.bar(range_,average, color="b", yerr=std, alpha=0.4,
align="center")
plt.xticks(range_, range_)
plt.show()
```



It is important to note the use of the `plt.xticks` command (and `plt.yticks` for the ordinate axis). The first parameter informs the command about the number of ticks that have to be placed on the axis, and the second one explicates the labels that have to be put on the ticks.

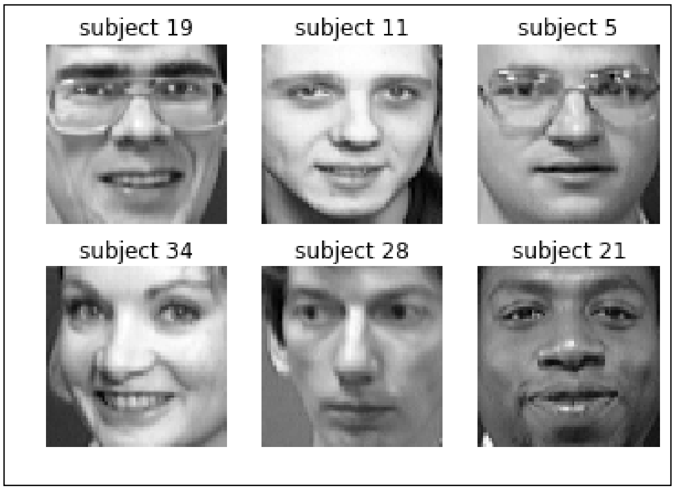
Another interesting parameter to notice is `alpha`, which has been used to set the transparency level of the bar. The `alpha` parameter is a float number ranging from 0.0, fully transparent, to 1.0, which is a solid color.

Image visualization

The last possible visualization is with regards to images. Using `plt.imgshow` is useful when you are working with image data. Let's take as an example the Olivetti dataset, an open source set of images of 40 people who provided 10 images of themselves at different times and with different expressions. The images are provided as features vectors of pixel intensities. So, it is important to reshape the vectors in order to make them resemble a matrix of pixels. Setting the interpolation to `'nearest'` helps you smoothen the picture:

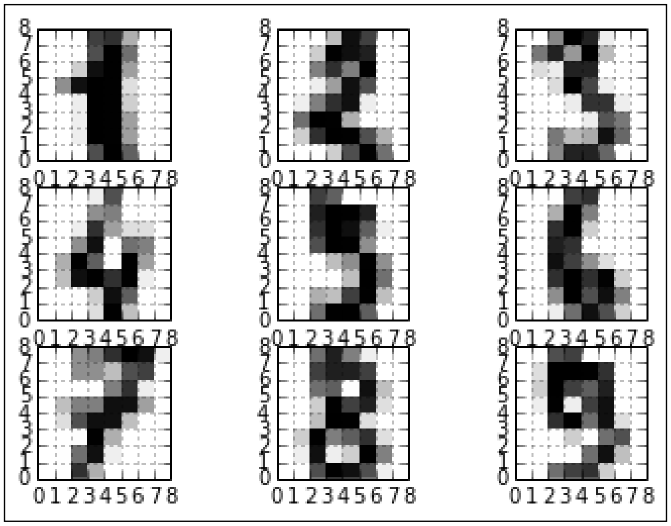
```
In: from sklearn.datasets import fetch_olivetti_faces
import numpy as np
import matplotlib.pyplot as plt
dataset = fetch_olivetti_faces(shuffle=True, random_state=5)
photo = 1
```

```
for k in range(6):
    subplot(2,3,k)
    plt.imshow(dataset.data[k].reshape(64,64), cmap=plt.cm.gray,
interpolation='nearest')
    plt.title('subject '+str(dataset.target[k]))
    plt.axis('off')
plt.show()
```



Visualization can be applied to handwritten digit or writing recognition. We will plot the first nine digits from the Scikit-learn handwritten digit dataset and set the extent of both the axes (by using the `extent` parameter and providing a list of minimum and maximum values) to align the grid to the pixels:

```
In: from sklearn.datasets import load_digits
digits = load_digits()
for number in range(1,10):
      plt.subplot(3, 3, number)
      plt.imshow(digits.images[number],
            cmap='binary',interpolation='none', extent=[0,8,0,8])
      # Extent defines the images max and min of the horizontal and
vertical values
      plt.grid()
plt.show()
```



Selected graphical examples with pandas

Though many machine learning algorithms, with appropriately set hyper-parameters, can optimally learn how to map your data with respect to your target outcome, their performance can be further improved by knowing about hidden and subtle problems in data. It is not simply a matter of detecting any missing or outlying case. Sometimes, it is paramount to clarify whether there are any groups or unusual distributions in the data(for instance, multimodal distributions). Clear data plots that explicate the relationship between variables can also lead to the creation of newer and better features that can predict more than the existing ones.

The practice that was just described is called Explorative Data Analysis, which can be effective if it has the following characteristics:

- It should be fast, allowing you to explore, develop new ideas and test them, and restart with a new exploration and fresh ideas.
- It should be graphic in order to better represent data as a whole, no matter how high its dimensionality is.

The pandas DataFrame offers many EDA tools that can help you in your explorations. However, you have to first place your data into a DataFrame:

```
In: import pandas as pd
print  'Your pandas version is: %s' % pd.__version__
from sklearn.datasets import load_iris
```

```
iris = load_iris()
iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
groups = list(iris.target)
iris_df['groups'] = pd.Series([iris.target_names[k] for k in groups])
Out: Your pandas version is: 0.15.2
```

> Check your pandas version. We tested this code under the 0.15.2 version of pandas, and it should also hold for the following releases.

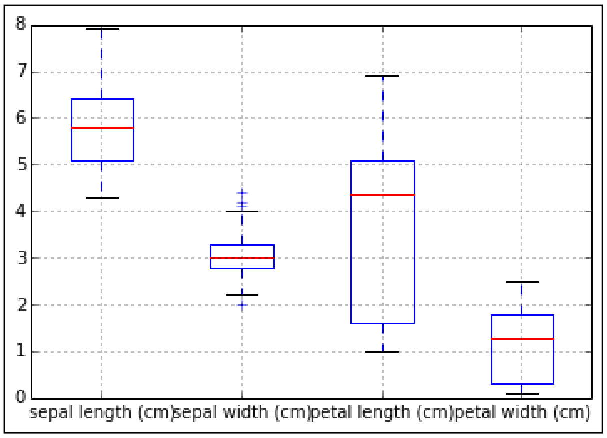
We will be using the iris_df DataFrame for all the examples presented in the following paragraphs.

pandas actually relies on matplotlib functions for its visualizations. It simply provides a convenient wrapper around the otherwise complex plotting instructions. This offers advantages in terms of speed and simplicity, which are the core values of any EDA process. Instead, if your purpose is to best communicate the findings by using beautiful visualization, you may notice that it is not so easy to customize the pandas graphical outputs. Therefore, when it is paramount to create specific graphic outputs, it is better to start working directly from the beginning with matplotlib instructions.

Boxplots and histograms

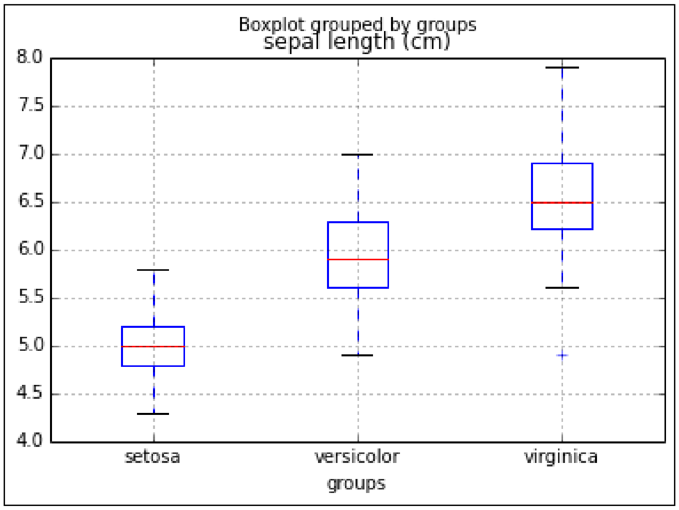
Distributions should always be the first aspect to be checked in your data. Boxplots draft the key figures in the distribution and help you spot outliers. Just use the boxplot method on your DataFrame for a quick overview:

```
In: boxplots = iris_df.boxplot(return_type='axes')
```



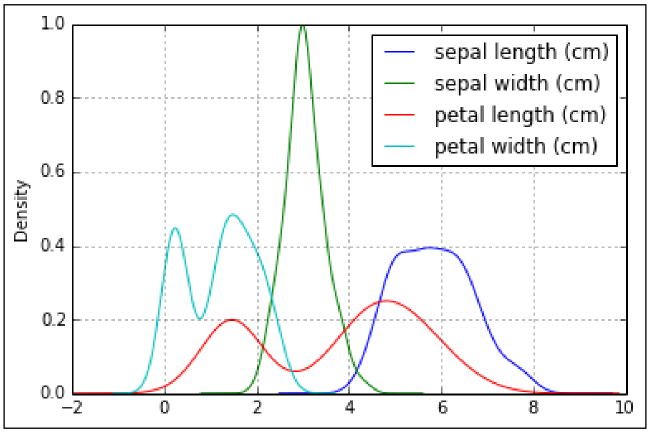
If you already have groups in your data (from categorical variables, or they may be derived from unsupervised learning), just point out the variable for which you need the boxplot and specify that you need to have the data separated by the groups (use the `by` parameter followed by the string name of the grouping variable):

```
In: boxplots = iris_df.boxplot(column='sepal length (cm)',
by='groups', return_type='axes')
```

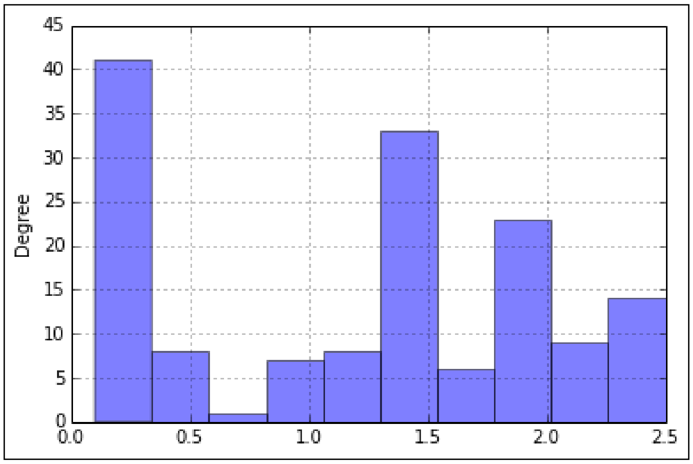


In this way, you can quickly know whether the variable is a good discriminator of the group differences. Anyway, Boxplots cannot provide you with a complete view of distributions as histograms and density plots. For instance, by using histograms and density plots, you can figure out whether there are distribution peaks or valleys:

```
In: densityplot = iris_df.plot(kind='density')
```



```
single_distribution = iris_df['petal width (cm)'
].plot(kind='hist', alpha=0.5)
```

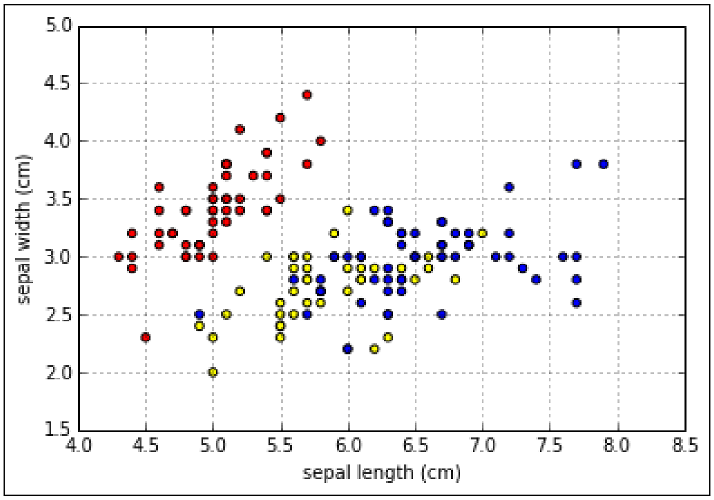


You can obtain both histograms and density plots by using the `plot` method. This method allows you to represent the whole dataset, specific groups of variables (you just have to provide a list of the string names and do some fancy indexing), or even single variables.

Scatterplots

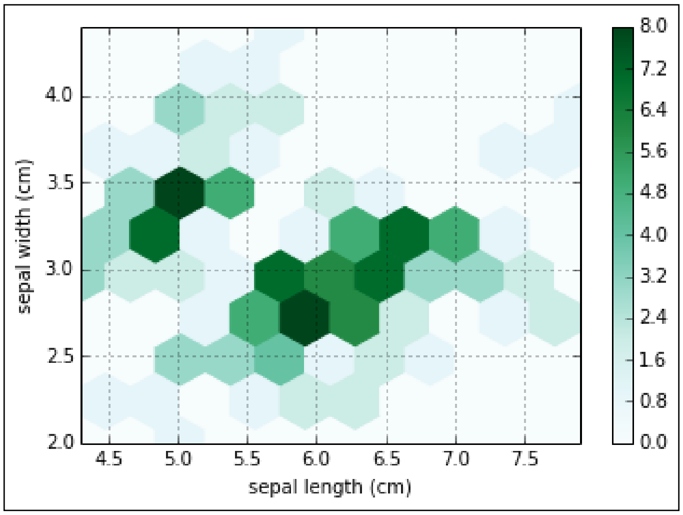
Scatterplots can be used to effectively understand whether the variables are in a nonlinear relationship, and you can get an idea about their best possible transformations to achieve linearization. If you are using an algorithm based on linear combinations, such as linear or logistic regression, figuring out how to render their relationship more linearly will help you achieve a better predictive power:

```
In: colors_palette = {0: 'red', 1: 'yellow', 2:'blue'}
colors = [colors_palette[c] for c in groups]
simple_scatterplot = iris_df.plot(kind='scatter', x=0, y=1, c=colors)
```



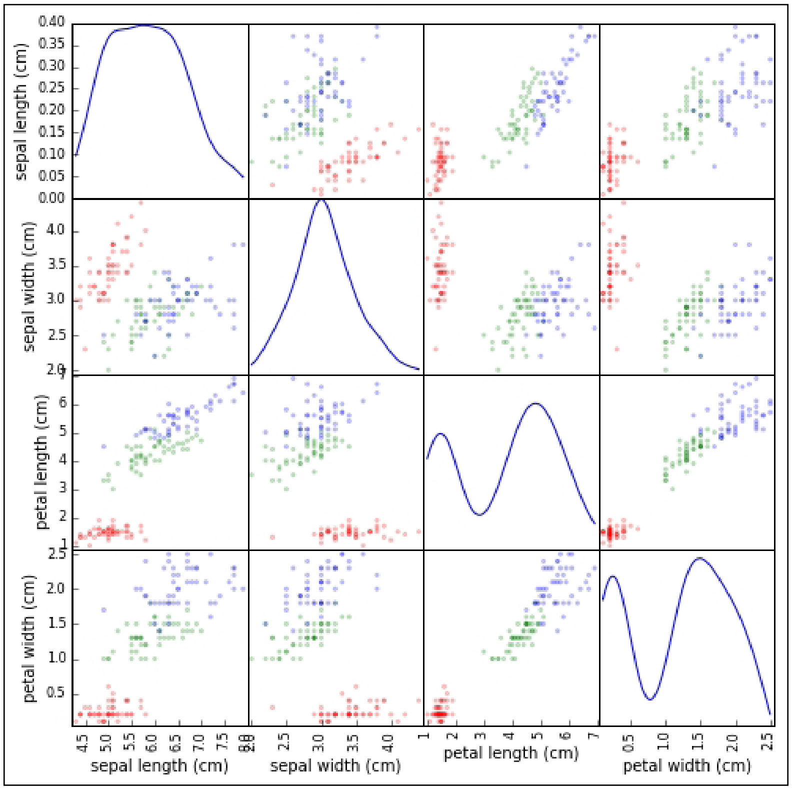
Scatterplots can be turned into hexagonal binning plots. Also, they help you effectively visualize the point densities, thus revealing natural clusters hidden in your data by using some of the variables in the dataset or the dimensions obtained by PCA or other dimensionality reduction algorithm:

```
In: hexbin = iris_df.plot(kind='hexbin', x=0, y=1, gridsize=10)
```



Scatterplots are bivariate. So, you'll require a single plot for every variable combination. If your variables are less in number (otherwise, the visualization will get cluttered), a quick turnaround is to automatically place a command to draw a matrix of scatterplots:

```
In: from pandas.tools.plotting import scatter_matrix
colors_palette = {0: "red", 1: "green", 2: "blue"}
colors = [colors_palette[c] for c in groups]
matrix_of_scatterplots = scatter_matrix(iris_df, alpha=0.2,
figsize=(6, 6), color=colors, diagonal='kde')
```

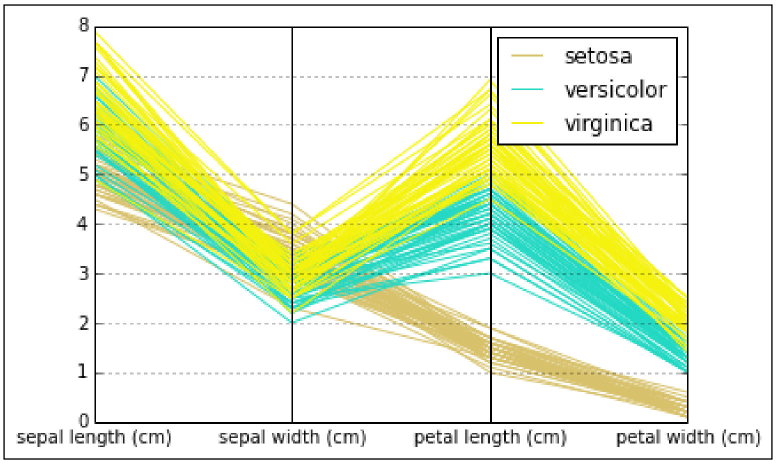


There are a few parameters that can control the various aspects of the scatterplot matrix. The `alpha` parameter controls the amount of transparency, and `figsize` provides the width and height of the matrix in inches. Finally, `color` accepts a list indicating the color of each point in the plot, thus allowing the depicting of different groups in data. Also, by selecting `'kde'` or `'hist'` on your `diagonal` parameter, you can opt to represent density curves or histograms (faster) of each variable on the diagonal of the scatter matrix.

Parallel coordinates

The scatterplot matrix can inform you about the conjoint distributions of your features. It thus helps you locate groups in data and verify their separability. Another useful plot that is helpful in the task of providing you with a hint about the most group-discriminating variables is parallel coordinates. By plotting all the observations as parallel lines with respect to all the possible variables (arbitrarily aligned on the abscissa), parallel coordinates will help you spot whether there are streams of observations grouped as your classes and understand the variables that best separate the streams (the most useful predictor variables):

```
In: from pandas.tools.plotting import parallel_coordinates
pll = parallel_coordinates(iris_df,'groups')
```



The `parallel_coordinates` is a pandas function and, to work properly, it just needs as parameters the `data` DataFrame and the string name of the variable containing the groups whose separability you want to test. This is why you should add it to your dataset. However, don't forget to remove it after you finish exploring by using the `DataFrame.drop('variable name',axis=1)` method.

Advanced data learning representation

Some useful representations can be derived. This is not done directly from the data; it is achieved by using machine learning procedures, which inform us about how the algorithms operate and offer us a more precise overview of the role of each predictor in the predictions obtained. In particular, learning curves can provide a quick diagnosis to improve your models. It helps you figure out whether you need more observations, or you need to enrich your variables.

Learning curves

A learning curve is a useful diagnostic graphic that depicts the behavior of your machine learning algorithm (your hypothesis) with respect to the available quantity of observations. The idea is to compare how the training performance (the error or accuracy of the in-sample cases) behaves with respect to the cross-validation (usually ten-fold) using different in-sample sizes.

As far as the training performance is concerned, you should expect it to be high at the start and then decrease. However, depending on the bias and variance level of the hypothesis, you will notice different behaviors:

- High bias hypothesis tends to start with average performances, decreases rapidly on being exposed to more complex data, and then remains at the same level of performance, no matter how many cases you further add. Low bias learners tend to generalize better in presence of many cases, but they are limited in their capability to approximate complex data structures, and hence their limited performance.
- High variance hypothesis tends to start high in performance and then slowly decreases as you add more cases. It tends to decrease slowly because it has a high capacity of recording the in-sample characteristics.

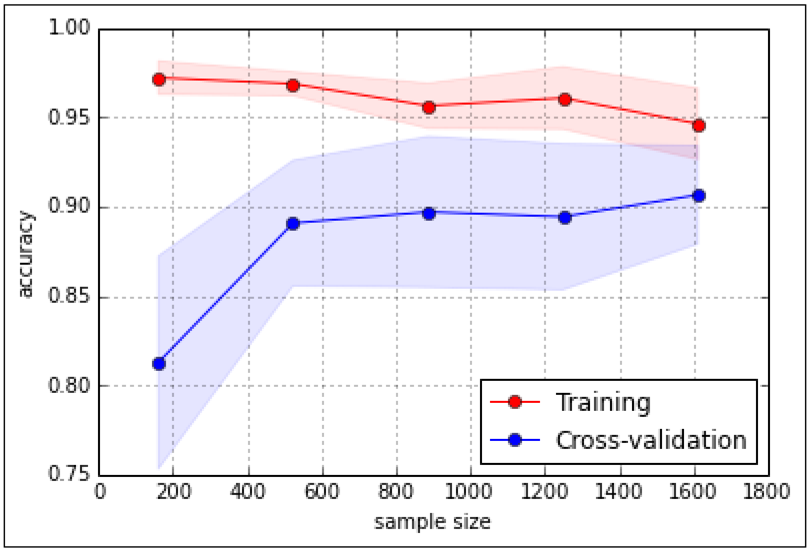
As for cross-validation, we can notice two behaviors:

- High bias hypothesis tends to start with low performance, but it grows very rapidly until it reaches almost the same performance as that of the training. Then, it stops growing.
- High variance hypothesis tends to start with very low performance. Then, steadily but slowly, it improves as more cases help generalize. It hardly reads the in-sample performances, and there is always a gap between them.

Being able to immediately estimate whether your machine learning solution is behaving as a high bias or high variance hypothesis helps you decide how to improve it. Scikit-learn makes it simpler to calculate all the statistics that are necessary for the drawing of the visualization thanks to the `learning_curve` class, though drawing them requires a few further calculations and commands:

```
In: import numpy as np
from sklearn.learning_curve import learning_curve, validation_curve
from sklearn.datasets import load_digits
from sklearn.linear_model import SGDClassifier
digits = load_digits()
X, y = digits.data, digits.target
```

```
hypothesis = SGDClassifier(loss='log', shuffle=True, n_iter=5,
penalty='l2', alpha=0.0001, random_state=3)
train_size, train_scores, test_scores = learning_curve(hypothesis, X,
      y, train_sizes=np.linspace(0.1,1.0,5), cv=10,
      scoring='accuracy', exploit_incremental_learning=False,
      n_jobs=-1)
mean_train  = np.mean(train_scores,axis=1)
upper_train = np.clip(mean_train + np.std(train_scores,axis=1),0,1)
lower_train = np.clip(mean_train - np.std(train_scores,axis=1),0,1)
mean_test   = np.mean(test_scores,axis=1)
upper_test = np.clip(mean_test + np.std(test_scores,axis=1),0,1)
lower_test = np.clip(mean_test - np.std(test_scores,axis=1),0,1)
plt.plot(train_size,mean_train,'ro-', label='Training')
plt.fill_between(train_size, upper_train, lower_train, alpha=0.1,
      color='r')
plt.plot(train_size,mean_test,'bo-', label='Cross-validation')
plt.fill_between(train_size, upper_test, lower_test, alpha=0.1,
      color='b')
plt.grid()
plt.xlabel('sample size') # adds label to x axis
plt.ylabel('accuracy') # adds label to y axis
plt.legend(loc='lower right', numpoints= 1)
plt.show()
```



The `learning_curve` class requires the following as an input:

- A series of training sizes stored in a list
- An indication of the number of folds to use and the error measure
- Your machine learning algorithm to test (parameter estimator)
- The predictors (parameter `X`) and the target outcome (parameter `y`)

As a result, the class will produce three arrays—the first one containing the effective training sizes, the second having the training scores for each cross-validation iteration, and the last one having the cross-validation scores.

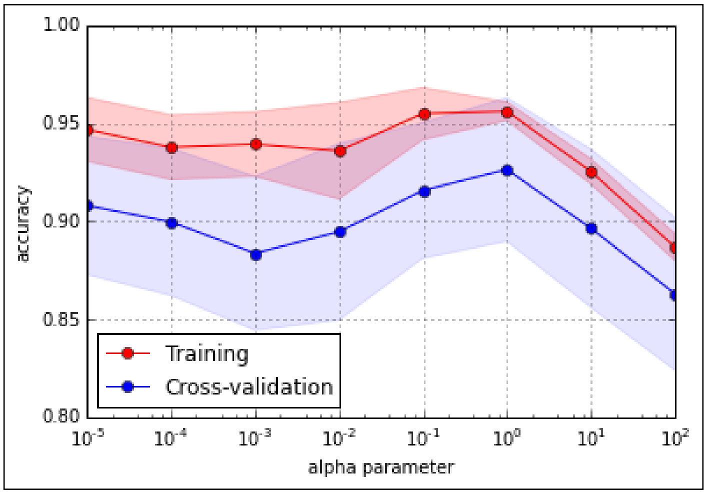
By applying the mean and the standard deviation for both training and cross-validation, it is possible to display in the graph both the curve trends and their variation. You can also provide information about the stability of the recorded performances.

Validation curves

As learning curves operate on different sample sizes, validation curves estimate the training and cross-validation performance with respect to the values that a hyper-parameter can take. As in learning curves, similar considerations can be made, though this visualization will grant you further insight about the optimization behavior of your parameter, visually suggesting to you the part of the hyper-parameter space that you should concentrate your search on:

```
In: from sklearn.learning_curve import validation_curve
testing_range = np.logspace(-5,2,8)
hypothesis = SGDClassifier(loss='log', shuffle=True, n_iter=5,
penalty='l2', alpha=0.0001, random_state=3)
train_scores, test_scores = validation_curve(hypothesis, X, y,
param_name='alpha', param_range=testing_range, cv=10,
scoring='accuracy', n_jobs=-1)
mean_train  = np.mean(train_scores,axis=1)
upper_train = np.clip(mean_train + np.std(train_scores,axis=1),0,1)
lower_train = np.clip(mean_train - np.std(train_scores,axis=1),0,1)
mean_test   = np.mean(test_scores,axis=1)
upper_test = np.clip(mean_test + np.std(test_scores,axis=1),0,1)
lower_test = np.clip(mean_test - np.std(test_scores,axis=1),0,1)
plt.semilogx(testing_range,mean_train,'ro-', label='Training')
plt.fill_between(testing_range, upper_train, lower_train, alpha=0.1,
color='r')
```

```
plt.fill_between(testing_range, upper_train, lower_train, alpha=0.1,
color='r')
plt.semilogx(testing_range,mean_test,'bo-', label='Cross-validation')
plt.fill_between(testing_range, upper_test, lower_test, alpha=0.1,
color='b')
plt.grid()
plt.xlabel('alpha parameter') # adds label to x axis
plt.ylabel('accuracy') # adds label to y axis
plt.ylim(0.8,1.0)
plt.legend(loc='lower left', numpoints= 1)
plt.show()
```



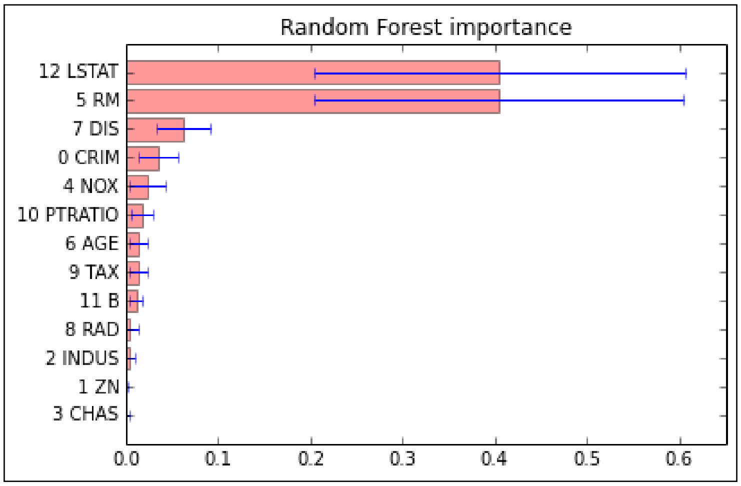
The syntax of the `validation_curve` class is similar to the previously seen `learning_curve` but for the `param_name` and `param_range` parameters, which should be respectively provided with the hyper-parameter and the range of the same that has to be tested. As for the results, the training and test results are provided in arrays.

Feature importance

As discussed in the conclusion of *Chapter 3*, *The Data Science Pipeline*, selecting the right variables can improve your learning process by reducing noise, variance of estimates, and the burden of too many computations. Ensemble methods, such as RandomForests in particular, can provide you with a different view of the role played by a variable when working together with other ones in your dataset compared to the greedy approach of a stepwise backward or forward selection.

In a few steps, we'll learn how to obtain such information and project it on a clear visualization:

```
In: from sklearn.datasets import load_boston
boston = load_boston()
X, y = boston.data, boston.target
feature_names = np.array([' '.join([str(b), a]) for a,b in
zip(boston.feature_names,range(len(boston.feature_names)))])
from sklearn.ensemble import RandomForestRegressor
RF = RandomForestRegressor(n_estimators=100, random_state=101).fit(X,
y)
importance = np.mean([tree.feature_importances_ for tree in
RF.estimators_],axis=0)
std = np.std([tree.feature_importances_ for tree in
RF.estimators_],axis=0)
indices = np.argsort(importance)
range_ = range(len(importance))
plt.figure()
plt.title("Random Forest importance")
plt.barh(range_,importance[indices],
       color="r", xerr=std[indices], alpha=0.4, align="center")
plt.yticks(range(len(importance)), feature_names[indices])
plt.ylim([-1, len(importance)])
plt.xlim([0.0, 0.65])
plt.show()
```



For each of the estimators (in our case, 100), the algorithm estimated a score to rank each variable importance. The RandomForest model is made up of complex decision trees that are made up of many branches. One of its variables is deemed important if, after casually permuting its original values, the resulting predictions of the permuted model are very different in terms of accuracy as compared to the predictions of the original model.

The importance vectors are averaged over the number of estimators, and the standard deviation of the estimations is computed by a list comprehension (the assignment of variables importance and `std`). Now, sorted according to the importance score (the vector indices), the results are projected onto a bar graph with an error bar provided by the standard deviation.

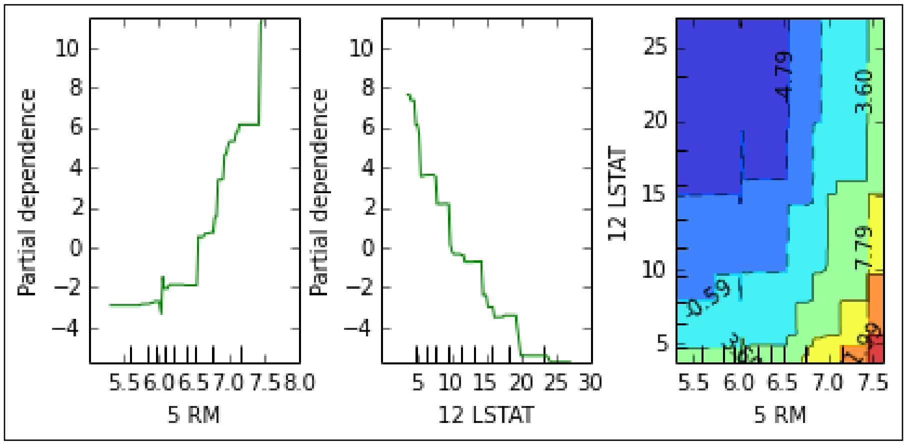
In our LSTAT analysis, the percentage of lower status population in the area and RM, which is the average number of rooms per dwelling, are pointed out as the most decisive variables in our RandomForest model.

GBT partial dependence plot

The estimate of the importance of a feature is a piece of information that can help you operate on the best choices to determine the features to be used. Sometimes, you may need to understand better why a variable is important in predicting a certain outcome. Gradient Boosting Trees, by controlling the effect of all the other variables involved in the analysis, provide you with a clear point of view of the relationship of a variable with respect to the predicted results. Such information can provide you with more insights about causation dynamics than what you may have obtained by using a very effective EDA:

```
In: from sklearn.ensemble.partial_dependence import
plot_partial_dependence

from sklearn.ensemble import GradientBoostingRegressor

GBM = GradientBoostingRegressor(n_estimators=100,
random_state=101).fit(X, y)

features = [5,12,(5,12)]
```

```
fig, axis = plot_partial_dependence(GBM, X, features,
feature_names=feature_names)
```



The plot_partial_dependence class will automatically provide you with the visualization after you provide an analysis plan on your part. You need to present a list of indexes of the features to be plotted singularly and the tuples of the indexes of those that you would like to plot on a heat map (the features are the axis, and the heat value corresponds to the outcome).

In the preceding example, both the average number of rooms and the percentage of lower status population have been represented, thus displaying an expected behavior. Interestingly, the heat map, which explains how they together contribute to the value of the outcome, reveals that they do not interact in any particular way (it is a single hill-climbing). However, it is also revealed that LSTAT is a strong delimiter of the resulting housing values when it is above 5.

Summary

This chapter provides an overview of essential data science by providing examples of both basic and advanced graphical representations of data, machine learning processes, and results. We explored the pylab module from matplotlib, which is the easiest and fastest access to the graphical capabilities of the package, used pandas for EDA, and tested the graphical utilities provided by Scikit-learn. All examples were like building blocks, and they are all easily customizable in order to provide you with a fast template for visualization.

In conclusion, this book covered all the key points of a data science project, presenting you with all the essential tools to operate your own projects using Python. As a learning tool, the book accompanied you through all the phases of data science, from data loading to machine learning and visualization, illustrating best practices and ways to avoid common pitfalls. As a reference, the book touched upon a variety of commands and packages, providing you with simple, clear instructions and examples that you can count on and which can make you save tons of time during your work.

From here on, Python will surely play a major role in your project developments, and we were glad to accompany you in your path towards the mastering of Python data science.

Index

[PACKT] open source
PUBLISHING community experience distilled

Thank you for buying
Python Data Science Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

About Packt Open Source

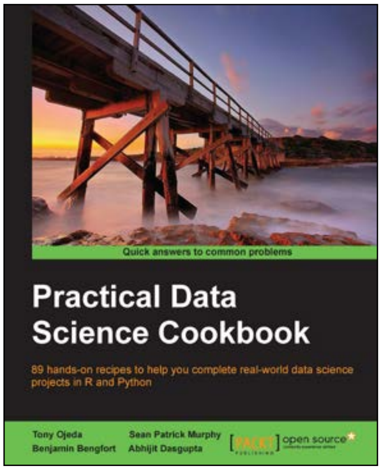
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
PUBLISHING community experience distilled

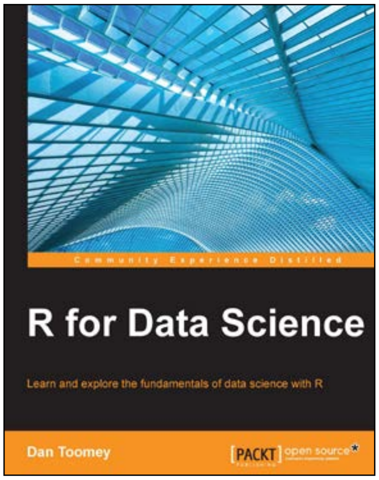

Quick answers to common problems
Practical Data Science Cookbook
89 hands-on recipes to help you complete real-world data science projects in R and Python
Tony Ojeda
Benjamin Bengfort
Sean Patrick Murphy
Abhijit Dasgupta
[PACKT] open source

Practical Data Science Cookbook

ISBN: 978-1-78398-024-6 Paperback: 396 pages

89 hands-on recipes to help you complete real-world data science projects in R and Python

1. Learn about the data science pipeline and use it to acquire, clean, analyze, and visualize data.
2. Understand critical concepts in data science in the context of multiple projects.
3. Expand your numerical programming skills through step-by-step code examples and learn more about the robust features of R and Python.


Community Experience Distilled
R for Data Science
Learn and explore the fundamentals of data science with R
Dan Toomey
[PACKT] open source

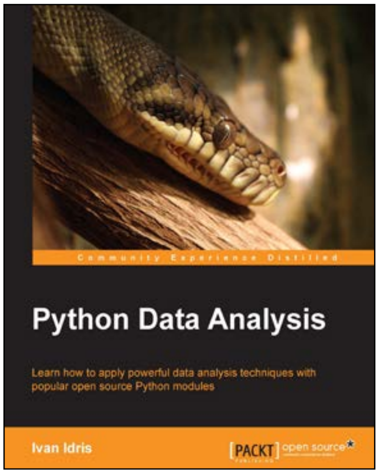
R for Data Science

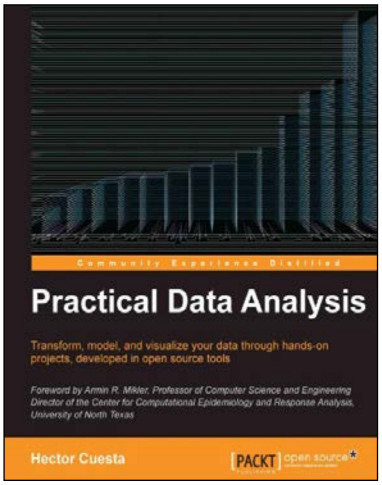
ISBN: 978-1-78439-086-0 Paperback: 364 pages

Learn and explore the fundamentals of data science with R

1. Familiarize yourself with R programming packages and learn how to utilize them effectively.
2. Learn how to detect different types of data mining sequences.
3. A step-by-step guide to understanding R scripts and the ramifications of your changes.

Please check **www.PacktPub.com** for information on our titles


Community Experience Distilled
Python Data Analysis
Learn how to apply powerful data analysis techniques with popular open source Python modules
Ivan Idris
[PACKT] open source


Community Experience Distilled
Practical Data Analysis
Transform, model, and visualize your data through hands-on projects, developed in open source tools
Foreword by Armin R. Mikler, Professor of Computer Science and Engineering Director of the Center for Computational Epidemiology and Response Analysis, University of North Texas
Hector Cuesta
[PACKT] open source