



Quick answers to common problems

Python Data Science Cookbook

Over 60 practical recipes to help you explore Python and its robust data science capabilities

Gopi Subramanian

[PACKT] open source[®]
PUBLISHING community experience distilled

Python Data Science Cookbook

Over 60 practical recipes to help you explore Python and its robust data science capabilities

Gopi Subramanian



open source community experience distilled

BIRMINGHAM - MUMBAI

Python Data Science Cookbook

Copyright © 2015 Packt Publishing

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

Every effort has been made in the preparation of this 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 author 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: November 2015

Production reference: 1041115

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

ISBN 978-1-78439-640-4

www.packtpub.com

Credits

Author

Gopi Subramanian

Project Coordinator

Kranti Berde

Reviewer

Bastiaan Sjardin

Proofreader

Safis Editing

Commissioning Editor

Akram Hussain

Indexer

Mariammal Chettiar

Acquisition Editor

Nikhil Karkal

Graphics

Disha Haria

Content Development Editor

Siddhesh Salvi

Production Coordinator

Nilesh Mohite

Technical Editor

Danish Shaikh

Cover Work

Nilesh Mohite

Copy Editor

Tasneem Fatehi

About the Author

Gopi Subramanian is a data scientist with over 15 years of experience in the field of data mining and machine learning. During the past decade, he has designed, conceived, developed, and led data mining, text mining, natural language processing, information extraction and retrieval, and search systems for various domains and business verticals, including engineering infrastructure, consumer finance, healthcare, and materials. In the loyalty domain, he has conceived and built innovative consumer loyalty models and designed enterprise-wide systems for personalized promotions. He has filed over ten patent applications at the US and Indian patent office and has several publications to his credit. He currently lives and works in Bangalore, India.

About the Reviewer

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 from 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 at Coursera in the social network analysis course from the University of Michigan and the practical machine learning course from Johns Hopkins University. His programming language of choice is R and Python. Currently, he is the cofounder of Quandbee (www.quandbee.com), a company specializing in machine learning applications.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print, and bookmark content
- ▶ On demand and accessible via a web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	v
Chapter 1: Python for Data Science	1
Introduction	2
Using dictionary objects	2
Working with a dictionary of dictionaries	6
Working with tuples	7
Using sets	12
Writing a list	15
Creating a list from another list - list comprehension	19
Using iterators	22
Generating an iterator and a generator	24
Using iterables	26
Passing a function as a variable	27
Embedding functions in another function	28
Passing a function as a parameter	29
Returning a function	30
Altering the function behavior with decorators	31
Creating anonymous functions with lambda	34
Using the map function	35
Working with filters	36
Using zip and izip	37
Processing arrays from the tabular data	39
Preprocessing the columns	43
Sorting lists	45
Sorting with a key	46
Working with itertools	51

Table of Contents

Chapter 2: Python Environments	55
Introduction	55
Using NumPy libraries	55
Plotting with matplotlib	65
Machine learning with scikit-learn	75
Chapter 3: Data Analysis – Explore and Wrangle	85
Introduction	86
Analyzing univariate data graphically	87
Grouping the data and using dot plots	95
Using scatter plots for multivariate data	100
Using heat maps	104
Performing summary statistics and plots	109
Using a box-and-whisker plot	114
Imputing the data	117
Performing random sampling	120
Scaling the data	122
Standardizing the data	124
Performing tokenization	127
Removing stop words	131
Stemming the words	135
Performing word lemmatization	138
Representing the text as a bag of words	140
Calculating term frequencies and inverse document frequencies	146
Chapter 4: Data Analysis – Deep Dive	151
Introduction	151
Extracting the principal components	153
Using Kernel PCA	160
Extracting features using singular value decomposition	166
Reducing the data dimension with random projection	171
Decomposing the feature matrices using non-negative matrix factorization	175
Chapter 5: Data Mining – Needle in a Haystack	185
Introduction	185
Working with distance measures	186
Learning and using kernel methods	192
Clustering data using the k-means method	196
Learning vector quantization	202
Finding outliers in univariate data	208
Discovering outliers using the local outlier factor method	216

Table of Contents

Chapter 6: Machine Learning 1	227
Introduction	227
Preparing data for model building	228
Finding the nearest neighbors	234
Classifying documents using Naïve Bayes	242
Building decision trees to solve multiclass problems	255
Chapter 7: Machine Learning 2	267
Introduction	267
Predicting real-valued numbers using regression	268
Learning regression with L2 shrinkage – ridge	283
Learning regression with L1 shrinkage – LASSO	293
Using cross-validation iterators with L1 and L2 shrinkage	301
Chapter 8: Ensemble Methods	315
Introduction	315
Understanding Ensemble – Bagging Method	317
Understanding Ensemble – Boosting Method	325
Understanding Ensemble – Gradient Boosting	341
Chapter 9: Growing Trees	357
Introduction	357
Going from trees to Forest – Random Forest	358
Growing Extremely Randomized Trees	369
Growing Rotational Forest	376
Chapter 10: Large-Scale Machine Learning – Online Learning	387
Introduction	387
Using perceptron as an online learning algorithm	388
Using stochastic gradient descent for regression	396
Using stochastic gradient descent for classification	405
Index	411

Preface

Today, we live in a world of connected things where tons of data is generated and it is humanly impossible to analyze all the incoming data and make decisions. Human decisions are increasingly replaced by decisions made by computers. Thanks to the field of data science. Data science has penetrated deeply in our connected world and there is a growing demand in the market for people who not only understand data science algorithms thoroughly, but are also capable of programming these algorithms. Data science is a field that is at the intersection of many fields, including data mining, machine learning, and statistics, to name a few. This puts an immense burden on all levels of data scientists; from the one who is aspiring to become a data scientist and those who are currently practitioners in this field. Treating these algorithms as a black box and using them in decision-making systems will lead to counterproductive results. With tons of algorithms and innumerable problems out there, it requires a good grasp of the underlying algorithms in order to choose the best one for any given problem.

Python as a programming language has evolved over the years and today, it is the number one choice for a data scientist. Its ability to act as a scripting language for quick prototype building and its sophisticated language constructs for full-fledged software development combined with its fantastic library support for numeric computations has led to its current popularity among data scientists and the general scientific programming community. Not just that, Python is also popular among web developers; thanks to frameworks such as Django and Flask.

This book has been carefully written to cater to the needs of a diverse range of data scientists—starting from novice data scientists to experienced ones—through carefully crafted recipes, which touch upon the different aspects of data science, including data exploration, data analysis and mining, machine learning, and large scale machine learning. Each chapter has been carefully crafted with recipes exploring these aspects. Sufficient math has been provided for the readers to understand the functioning of the algorithms in depth. Wherever necessary, enough references are provided for the curious readers. The recipes are written in such a way that they are easy to follow and understand.

This book brings the art of data science with power Python programming to the readers and helps them master the concepts of data science. Knowledge of Python is not mandatory to follow this book. Non-Python programmers can refer to the first chapter, which introduces the Python data structures and function programming concepts.

The early chapters cover the basics of data science and the later chapters are dedicated to advanced data science algorithms. State-of-the-art algorithms that are currently used in practice by leading data scientists across industries including the ensemble methods, random forest, regression with regularization, and others are covered in detail. Some of the algorithms that are popular in academia and still not widely introduced to the mainstream such as rotational forest are covered in detail.

With a lot of do-it-yourself books on data science today in the market, we feel that there is a gap in terms of covering the right mix of math philosophy behind the data science algorithms and implementation details. This book is an attempt to fill this gap. With each recipe, just enough math introductions are provided to contemplate how the algorithm works; I believe that the readers can take full benefits of these methods in their applications.

A word of caution though is that these recipes are written with the objective of explaining the data science algorithms to the reader. They have not been hard-tested in extreme conditions in order to be production ready. Production-ready data science code has to go through a rigorous engineering pipeline.

This book can be used both as a guide to learn data science methods and quick references. It is a self-contained book to introduce data science to a new reader with little programming background and help them become experts in this trade.

What this book covers

Chapter 1, Python for Data Science, introduces Python's built-in data structures and functions, which are very handy for data science programming.

Chapter 2, Python Environments, introduces Python's scientific programming and plotting libraries, including NumPy, matplotlib, and scikit-learn.

Chapter 3, Data Analysis – Explore and wrangle, covers data preprocessing and transformation routines to perform exploratory data analysis tasks in order to efficiently build data science algorithms.

Chapter 4, Data Analysis – Deep Dive, introduces the concept of dimensionality reduction in order to tackle the curse of dimensionality issues in data science. Starting with simple methods and moving on to the advanced state-of-the-art dimensionality reduction techniques are discussed in detail.

Chapter 5, Data Mining – Needle in a haystack Name, discusses unsupervised data mining techniques, starting with elaborate discussions on distance methods and kernel methods and following it up with clustering and outlier detection techniques.

Chapter 6, Machine Learning 1, covers supervised data mining techniques, including nearest neighbors, Naïve Bayes, and classification trees. In the beginning, we will lay a heavy emphasis on data preparation for supervised learning.

Chapter 7, Machine Learning 2, introduces regression problems and follows it up with topics on regularization including LASSO and ridge. Finally, we will discuss cross-validation techniques as a way to choose hyperparameters for these methods.

Chapter 8, Ensemble Methods, introduces various ensemble techniques including bagging, boosting, and gradient boosting. This chapter shows you how to make a powerful state-of-the-art method in data science where, instead of building a single model for a given problem, an ensemble or a bag of models are built.

Chapter 9, Growing Trees, introduces some more bagging methods based on tree-based algorithms. Due to their robustness to noise and universal applicability to a variety of problems, they are very popular among the data science community.

Chapter 10, Large scale machine learning – Online Learning covers large scale machine learning and algorithms suited to tackle such large scale problems. This includes algorithms that work with streaming data and data that cannot be fitted into memory completely.

What you need for this book

All the recipes in this book were developed and tested on an 8 GB machine with Intel i7 CPU running Windows 7 64-bit software.

Python 2.7.5, NumPy 1.8.0, SciPy 0.13.2, Matplotlib 1.3.1, NLTK 3.0.2, and scikit-learn 0.15.2 versions were used for the developing methods.

The same code should work on Linux variants and Macs with the appropriate libraries mentioned here. Alternatively, a Python virtual environment can be created with the version of these libraries and you can run all the recipes.

Who this book is for

This book is intended for all levels of data science professionals, both students and practitioners from novice to experts. Different recipes in the chapters cater to the needs of different audiences. Novice readers can spend some time in getting themselves acquainted with data science in the first five chapters. Experts can refer to the later chapters to refer/understand how advanced techniques are implemented using Python. The book covers just enough mathematics and provides the necessary references for computer programmers who wish to understand data science. People from a non-Python background can effectively use this book. The first chapter of the book introduces Python as a programming language for data science. It will be helpful if you have some prior basic programming experience. The book is mostly self-contained and introduces data science to a new reader and can help him become an expert in this trade.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

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

Code words in text, like function names are shown as follows:

We call `get_iris_data()` function to get the input data. We leverage the function `train_test_split` from Scikit learn's model `cross_validation` to split the input datasets into two.

A block of code is set as follows:

```
# Shuffle the dataset
shuff_index = np.random.shuffle(range(len(y)))
x_train = x[shuff_index, :].reshape(x.shape)
y_train = np.ravel(y[shuff_index, :])
```

Formulas are typically provided as images as follows,

$$x_i = \{x_{i1}, x_{i2}, \dots, x_{im}\} \text{ where } i = 1 \text{ to } n$$

Typically the math section is introduced at the beginning of each recipe. In some chapters the common math required for most of the recipes in that chapter are included in the introduction section of the first recipe.

External url's are specified as follows,

http://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html

Specific call-outs in some algorithm implementation details in a third party library is provided as follows.

'The predicted class of an input sample is computed as the class with the highest mean predicted probability. If base estimators do not implement a `predict_proba` method, then it resorts to voting.'

Where ever applicable references to scientific journals and papers are provided as follows,

Preface

Please refer to the paper by Leo Breiman for more information about bagging.

Leo Breiman. 1996. Bagging predictors. Mach. Learn. 24, 2 (August 1996), 123-140. DOI=10.1023/A:1018054314350 http://dx.doi.org/10.1023/A:1018054314350

Program output and graphs are typically provided as images. For example,

Single Model Accuracy on Dev data				
	precision	recall	f1-score	support
0	0.83	0.84	0.83	51
1	0.85	0.83	0.84	54
avg / total	0.84	0.84	0.84	105

Bagging Model Accuracy on Dev data				
	precision	recall	f1-score	support
0	0.85	0.88	0.87	51
1	0.88	0.85	0.87	54
avg / total	0.87	0.87	0.87	105

Any command-line input or output is written as follows:

```
Counter({'Peter': 4, 'of': 4, 'Piper': 4, 'pickled': 4, 'picked': 4, 'peppers': 4, 'peck': 4, 'a': 2, 'A': 1, 'the': 1, 'Wheres': 1, 'If': 1})
```

In places where we would like the reader to inspect some of the variables in Python shell, we specify it as follows,

```
>>> print b_tuple[0]
1
>>> print b_tuple[-1]
c
>>>
```



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.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from http://www.packtpub.com/sites/default/files/downloads/1234OT_ColorImages.pdf.

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

Python for Data Science

In this chapter, we will cover the following recipes:

- ▶ Using dictionary objects
- ▶ Working with a dictionary of dictionaries
- ▶ Working with tuples
- ▶ Using sets
- ▶ Writing a list
- ▶ Creating a list from another list - list comprehension
- ▶ Using iterators
- ▶ Generating an iterator and a generator
- ▶ Using iterables
- ▶ Passing a function as a variable
- ▶ Embedding functions in another function
- ▶ Passing a function as a parameter
- ▶ Returning a function
- ▶ Altering the function behavior with decorators
- ▶ Creating anonymous functions with lambda
- ▶ Using the map function
- ▶ Working with filters
- ▶ Using zip and izip
- ▶ Processing arrays from the tabular data
- ▶ Preprocessing the columns
- ▶ Sorting lists
- ▶ Sorting with a key
- ▶ Working with itertools

Introduction

The Python programming language provides a lot of built-in data structures and functions that are very handy for data science programming. In this chapter, we will look at some that are most frequently used. In the subsequent chapters, you will see that these will be used in various sections for different topics. A good grasp of these will help you in the long run to quickly bootstrap a program in order to handle data and develop algorithms.

While this chapter is a quick overview of handy data structures and methods, you will start discovering your own ways of combining these data structures to achieve your requirements as you become a seasoned Python user.

Each of these data structures have an utility, though different circumstances may require using two or more data structures in tandem to achieve your requirements. You will see this in action in some of the examples in this book.

Using dictionary objects

In Python, containers are objects that can hold any number of arbitrary objects. They provide a way to access the child objects and iterate over them. Dictionary, tuple, list, and set are container objects in Python. More container types are available with the `collections` module. Let's look at the dictionary object in detail in this section.

Getting ready

Let's look at an example Python script to understand how a dictionary operates. So, with a text, this script tries to get the word count, that is, how many times each word has appeared in the given text.

How to do it...

Let's proceed to demonstrate how to operate a dictionary in Python. Let's use a simple sentence to demonstrate the use of a dictionary. We will follow it up with an actual dictionary creation:

```
# 1.Load a variable with sentences
sentence = "Peter Piper picked a peck of pickled peppers A peck of
pickled \
peppers Peter Piper picked If Peter Piper picked a peck of pickled \
peppers Wheres the peck of pickled peppers Peter Piper picked"

# 2.Initialize a dictionary object
word_dict = {}
```

```
# 3. Perform the word count
for word in sentence.split():
    if word not in word_dict:
        word_dict[word] = 1
    else:
        word_dict[word] +=1
# 4. print the outputprint (word_dict)
```

How it works...

The preceding code builds a word frequency table; every word and its frequency is calculated. The final print statement produces the following output:

```
{'a': 2, 'A': 1, 'Peter': 4, 'of': 4, 'Piper': 4, 'pickled': 4,
'picked': 4, 'peppers': 4, 'the': 1, 'peck': 4, 'Wheres': 1, 'If': 1}
```

The preceding output is a key value pair. For each word (key), we have a frequency (value). A dictionary data structure is a hash map where values are stored against a key. In the preceding example, we used a string as a key; however, any other immutable data type can also be used as a key.

Refer to the following URL for a detailed discussion about mutable and immutable objections in Python:

<https://docs.python.org/2/reference/datamodel.html>

Similarly, values can be any data type including custom classes.

In step 2, we initialized the dictionary. Its empty when initialized. When a new key is added to a dictionary, accessing the dictionary through the new key will throw `KeyError`. In the preceding example in step 3, we included an if statement in the for loop to handle this situation. However, we can also use the following:

```
word_dict.setdefault(word, 0)
```

With every key access to the dictionary, this statement has to be repeated if we are adding elements to a dictionary in a loop, as in a loop, we are not aware of new keys. Rewriting step 3 using `setdefault` will look as follows:

```
for word in sentence.split():
    word_dict.setdefault(word, 0)
    word_dict[word] +=1
```

There's more...

Python 2.5 and above has a class named `defaultdict`; it's in the `collections` module. This takes care of the `setdefault` action. A `defaultdict` class is invoked as follows:

```
from collections import defaultdict

sentence = "Peter Piper picked a peck of pickled peppers A peck of
pickled \
           peppers Peter Piper picked If Peter Piper picked a peck of
pickled \
           peppers Wheres the peck of pickled peppers Peter Piper
picked"

word_dict = defaultdict(int)

for word in sentence.split():
    word_dict[word] += 1
print word_dict
```

As you have noticed, we included `collections.defaultdict` in our code and initialized our dictionary. Note that the `int` parameter, `defaultdict`, takes a function as an argument. In this case, we passed the `int()` function and thus, when the dictionary encounters a key that was not seen before, it initializes the key with a value returned by the `int()` function, in this case, zero. We will use `defaultdict` later in this book.



A typical dictionary does not remember the order in which the keys were inserted. In its `collections` module, Python provides a container called `OrderedDict` that can remember the order in which the keys were inserted. See the following Python documentation for more details:
<https://docs.python.org/2/library/collections.html#collections.OrderedDict>

Looping through a dictionary is very easy; using the `keys()` function provided in the dictionary, we can loop through the key and using `values()`, we can loop through the values or using `items()`, we can loop through both the keys and values. Look at the following example:

```
For key, value in word_dict.items():
    print key, value
```

In this example, using `dict.items()`, we can iterate through the keys and values present in the dictionary.

The Python documentation for dictionaries is very exhaustive and is a handy companion when working with dictionaries:

<https://docs.python.org/2/tutorial/datastructures.html#dictionaries>

Dictionaries are very useful as an intermediate data structure. If your program uses JSON as a way to move around information between modules, dictionary is the right data type for the job. It is very convenient to load a dictionary from a JSON file and similarly dump a dictionary as JSON strings.

Python provides us with libraries to handle JSON very efficiently:

<https://docs.python.org/2/library/json.html>

Counter is a dictionary subclass to count the hashable objects. Our example of the word count can be easily done using counter.

Look at the following example:

```
from collections import Counter

sentence = "Peter Piper picked a peck of pickled peppers A peck of
pickled \
           peppers Peter Piper picked If Peter Piper picked a peck of
pickled \
           peppers Wheres the peck of pickled peppers Peter Piper
picked"

words = sentence.split()

word_count = Counter(words)

print word_count['Peter']print word_dict
```

The output is as follows and you can verify this output with the previous one:

```
Counter({'Peter': 4, 'of': 4, 'Piper': 4, 'pickled': 4, 'picked': 4,
'peppers': 4, 'peck': 4, 'a': 2, 'A': 1, 'the': 1, 'Wheres': 1, 'If':
1})
```

You can go through the following link to understand more about Counters:

<https://docs.python.org/2/library/collections.html#collections.Counter>

See also

- ▶ *Working with Dictionary of Dictionaries* recipe in Chapter 1, Using Python for Data Science

Working with a dictionary of dictionaries

As we mentioned earlier, the real power of these data structures lies in how creatively you can use them to achieve your tasks. Let's look at an example to understand how to use dictionaries in a dictionary.

Getting ready

Look at the following table:

User/Movie	LOR1	LOR2	LOR3	SW1	SW2
Alice	4	5	3	5	3
Huntsman	1	2	1	4	4
Snipe	3	4	4	2	1

In the first column, we have three users and the rest of the columns are movies. The cell values are ratings given by a user for a movie. Let's say we want to represent this in memory so that some other part of a larger code base can easily access this information. We will use a dictionary of dictionaries to achieve this objective.

How to do it...

We will create the `user_movie_rating` dictionary using an anonymous function to demonstrate the concept of a dictionary of dictionaries.

We will fill it with data to show the effective use of a dictionary of dictionaries:

```
from collections import defaultdict

user_movie_rating = defaultdict(lambda :defaultdict(int))

# Initialize ratings for Alice
user_movie_rating["Alice"]["LOR1"] = 4
user_movie_rating["Alice"]["LOR2"] = 5
user_movie_rating["Alice"]["LOR3"] = 3
user_movie_rating["Alice"]["SW1"] = 5
user_movie_rating["Alice"]["SW2"] = 3
print user_movie_rating
```

How it works...

The `user_movie_rating` is a dictionary of dictionaries. As explained in the previous section, `defaultdict` takes a function for argument; in this case, we passed a built-in anonymous function, `lambda`, which returns a dictionary. So, every time a new key is passed to `user_movie_rating`, a new dictionary will be created for this key. We will see more about the `lambda` function in the subsequent section.

This way, we can access the rating of any user movie combination very quickly. Similarly, there are plenty of use cases where a dictionary of dictionaries comes in very handy.

As a closing note on the dictionary, I would like to mention that having a good grasp of the dictionary data structure will help ease a lot of your data science programming tasks. As we will see later, dictionaries are frequently used to store features and labels in machine learning. The Python NLTK library uses a dictionary extensively to store features in text mining:

<http://www.nltk.org/book/ch05.html>

The section titled *Mapping words to Properties using Python Dictionaries* is a good read to understand how effectively dictionaries can be used.

See also

- ▶ *Creating Anonymous Functions* recipe in Chapter 1, Using Python for Data Science

Working with tuples

A tuple is a type of container object known as sequence types in Python. Tuples are immutable and can have a heterogeneous sequence of elements separated by a comma and enclosed in parentheses. They support the following operations:

- ▶ `in` and `not in`
- ▶ Comparison, concatenation, slicing, and indexing
- ▶ `min()` and `max()`

Getting ready

Rather than having a full program as we did with dictionaries, we will see tuples as fragmented codes where we will concentrate on the creation and manipulation activities.

How to do it...

Let's see some scripts demonstrating the creation and manipulation of tuples:

```
# 1.Ways of creating a tuple
a_tuple = (1,2,'a')
b_tuple = 1,2,'c'

# 2.Accessing elements of a tuple through index
print b_tuple[0]
print b_tuple[-1]

# 3.It is not possible to change the value of an item in a tuple,
# for example the next statement will result in an error.
try:
    b_tuple[0] = 20
except:
    print "Cannot change value of tuple by index"

# 4.Though tuples are immutable
# But elements of a tuple can be mutable objects,
# for instance a list, as in the following line of code
c_tuple =(1,2,[10,20,30])
c_tuple[2][0] = 100

# 5.Tuples once created cannot be extended like list,
# however two tuples can be concatenated.

print a_tuple + b_tuple

# 6 Slicing of uples
a =(1,2,3,4,5,6,7,8,9,10)
print a[1:]
print a[1:3]
print a[1:6:2]
print a[::-1]

# 7.Tuple min max
print min(a),max(a)

# 8.in and not in
if 1 in a:
    print "Element 1 is available in tuple a"
else:
    print "Element 1 is available in tuple a"
```

How it works...

In step 1, we created a tuple. Though strictly speaking, the parentheses are not needed, still it's an option for better readability. As you can see, we created a heterogeneous tuple with numeric and string values. Step 2 details how the elements of a tuple can be accessed through the index. Indices start from zero. A negative number can be used to access the tuple in reverse. The output of the print statement is as follows:

```
>>> print b_tuple[0]
1
>>> print b_tuple[-1]
c
>>>
```

The Python tuple indices start from 0. Tuples are immutable.

In step 3, we will look at the most important property of a tuple called immutability. It is not possible to change the value of an item in a tuple; step 3 will result in an error thrown by the interpreter:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

This may look restrictive; however, the immutable feature has immense value from a data science perspective.



While building programs for machine learning, in particular during the feature generation from raw data, creating feature tuples ensures that values cannot be changed by downstream programs.

As these features reside in a tuple, no downstream program can accidentally change the feature values.

However, we want to point out that a tuple can have a mutable object as its member, for example, a list. If we have a tuple as shown in step 4, the third element of the tuple is a list. Now, let's try to change an element in the list:

```
c_tuple[2][0] = 100
```

We will print the tuple as follows:

```
print c_tuple
```

We will get the following output:

```
(1, 2, [100, 20, 30])
```

As you can see, the value of the first element in the list is changed to 100.

In step 5, we concatenated two tuples. Another interesting way to use tuples is when different modules are creating different features for a machine learning program.



For example, let's say that you have one module that is creating a bag-of-words kind of feature and another module that is working on creating numerical features for a typical text classification program. These models can output the tuples and a final module can concatenate these tuples to get a full feature vector.

Due to its immutable property, unlike a list, a tuple cannot be extended after its creation. It does not support an `append` function. Another advantage of this immutable property is that a tuple can be used as a key in a dictionary.



Typically, when creating keys, we may need to concatenate different string values using custom separators to create a unique key. Instead, a tuple with these string values can be created to be used as a key.

This improves the program output readability and also avoids bugs from creeping in when the keys are combined manually.

In step 6, we will detail the slicing operations in a tuple. Typically, there are three numbers provided for the slicing and they are separated by a colon. The first number decides which index to start the slicing, the second one decides the ending index, and the last one is for step. The examples in step 6 will clarify this:

```
print a[1:]
```

It prints the output as follows:

```
(2, 3, 4, 5, 6, 7, 8, 9, 10)
```

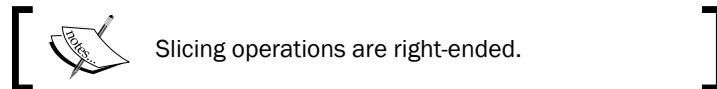
In this case, we specified only the start index number, 1. (Remember that indices start with zero.) We got a slice of the tuple starting from index, 1. Let's see another example:

```
print a[1:3]
```

It prints the output as follows:

```
(2, 3)
```

Here, we specified the start index as 1 and end index as 3.



Though we specified the end index as 3, the output will be returned till index 2, that is, one before. Hence, we have 2 and 3 as a part of our output slice. Finally, let's provide all three parameters and the start and end indices followed by the step size:

```
print a[1:6:2]
```

It displays the output as follows:

```
(2, 4, 6)
```

Here, our step size is 2. In addition to the start and end indices, we also specified the step size. Hence, it jumps two indices every time and produces the output as shown previously.

Let's look at the negative indexing:

```
print a[::-1]
```

Here, we used the negative index. The output is as follows:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Everything except the last element is returned in the slice:

```
print a[::-1]
```

Food for thought, the output of the preceding statement is as follows—a curious reader should be able to figure out how we got the following output:

```
(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
```

In step 7, we will show that we can use the `min()` and `max()` functions on a tuple to get the minimum and maximum value out of a tuple:

```
>>> print min(a), max(a)
1 10
>>>
```

In step 8, we will show the conditional operators in and not in; this can be effectively used to find out if an element is part of a tuple or not:

```
if 1 in a:
    print "Element 1 is available in tuple a"
else:
    print "Element 1 is not available in tuple a"
```

There's more...

As we saw in the preceding section, we accessed the elements of a tuple by their indices. For better program readability, say that we want to have a name assigned to each of the elements of a tuple and access the elements by their names. This is where namedtuple comes to our rescue. The following URL gives a good documentation of namedtuple:

<https://docs.python.org/2/library/collections.html#collections.namedtuple>

Let's look at a simple example to illustrate the use of a namedtuple:

```
from collections import namedtuple

vector = namedtuple("Dimension",'x y z')
vec_1 = vector(1,1,1)
vec_2 = vector(1,0,1)

manhattan_distance = abs(vec_1.x - vec_2.x) + abs(vec_1.y - vec_2.y) \
+ abs(vec_1.z - vec_2.z)

print "Manhattan distance between vectors = %d"%(manhattan_distance)
```

You can see that we accessed the elements of vec_1 and vec_2 using object notation, vec_1.x, vec_1.y, and so on. Instead of using their indices, we now have a better readable program. Vec_1.x is equivalent to vec_1[0].

See also

- ▶ *Data Analysis – Explore and Wrangle* recipe in Chapter 3, *Analyzing Data - Explore & Wrangle* which represents the text as a Bag-of-words.

Using sets

Sets are very similar to list data structures except that they do not allow duplicates. It's an unordered collection of homogeneous elements. Typically, sets are used to remove the duplicate elements from a list. However, a set supports operations such as intersection, union, difference, and symmetric difference. These operations are very handy in a lot of use cases.

Getting ready

In this section, we will write a small program to understand the various utilities of set data structures. In our example, we will calculate a similarity score between two sentences using Jaccard's coefficient. We will see in detail about Jaccard's coefficient and similar other measures in later chapters. Here is a quick introduction to this measure. Jaccard's coefficient is a number between zero and one, where one indicates a high similarity. It's calculated based on how many elements are common between the two sets.

How to do it...

Let's see some Python scripts that are used for the set creation and manipulation:

```
# 1. Initialize two sentences.  
st_1 = "dogs chase cats"  
st_2 = "dogs hate cats"  
  
# 2. Create set of words from strings  
st_1_wrds = set(st_1.split())  
st_2_wrds = set(st_2.split())  
  
# 3. Find out the number of unique words in each set, vocabulary size.  
no_wrds_st_1 = len(st_1_wrds)  
no_wrds_st_2 = len(st_2_wrds)  
  
# 4. Find out the list of common words between the two sets.  
# Also find out the count of common words.  
cmn_wrds = st_1_wrds.intersection(st_2_wrds)  
no_cmn_wrds = len(st_1_wrds.intersection(st_2_wrds))  
  
# 5. Get a list of unique words between the two sets.  
# Also find out the count of unique words.  
unq_wrds = st_1_wrds.union(st_2_wrds)  
no_unq_wrds = len(st_1_wrds.union(st_2_wrds))  
  
# 6. Calculate Jaccard similarity  
similarity = no_cmn_wrds / (1.0 * no_unq_wrds)  
  
# 7. Let us now print to grasp our output.  
print "No words in sent_1 = %d"%(no_wrds_st_1)  
print "Sentence 1 words =", st_1_wrds  
print "No words in sent_2 = %d"%(no_wrds_st_2)  
print "Sentence 2 words =", st_2_wrds  
print "No words in common = %d"%(no_cmn_wrds)
```

```
print "Common words =", cmn_wrds
print "Total unique words = %d"%(no_unq_wrds)
print "Unique words=", unq_wrds
print "Similarity = No words in common/No unique words, %d/%d =
%.2f"%(no_cmn_wrds,no_unq_wrds,similarity)
```

How it works...

In steps 1 and 2, we took two sentences, split them into words, and created two sets using the `set()` function. The `set` function can be used to convert a list or tuple to a set. Look at the following code snippet:

```
>>> a = (1, 2, 1)
>>> set(a)
set([1, 2])
>>> b = [1, 2, 1]
>>> set(b)
set([1, 2])
```

In this example, `a` is a tuple and `b` is a list. With the `set()` function, the duplicates are eliminated and a set object is returned. The `st_1.split()` and `st_2.split()` method return a list and we will pass it to a `set` function to get the set objects.

Let's now calculate the similarity score between two sentences using Jaccard's coefficient. We will see in detail about Jaccard's coefficient and similar other measures in the similarity measures section in a later chapter. We will leverage the `union()` and `intersection()` functions available with the sets to calculate the similarity score.

In step 4, we will perform two operations. First, using the `intersection()` function, we will try to find out what words are common between the sets. The common words between the two sentences are 'cats' and 'dogs'. Followed by this, we will find out the count of the common words, which is two. In the next step, we will find out the list of unique words between the two sets using the `union()` function. The unique words between these two sentences are 'cats', 'hate', 'dogs', and 'chase'. This is sometimes referred to as vocabulary in natural language processing. Finally, we will calculate Jaccard's coefficient in step 6, which is the ratio of a count of the common words between the two sets to a count of the unique words between the two sets.

The output of this program looks as follows:

```
No words in sent_1 = 3
Sentence 1 words = set(['cats', 'dogs', 'chase'])
No words in sent_2 = 3
Sentence 2 words = set(['cats', 'hate', 'dogs'])
No words in common = 2
Common words = set(['cats', 'dogs'])
```

```
Total unique words = 4
Unique words= set(['cats', 'hate', 'dogs', 'chase'])
Similarity = No words in common/No unique words, 2/4 = 0.50
```

There's more...

We gave the preceding example to represent the usage of the set functions. However, you can use the built-in functions from libraries such as scikit-learn. Going forward, we will leverage as much of these functions from libraries as possible, instead of hand coding these utility functions:

```
# Load libraries
from sklearn.metrics import jaccard_similarity_score

# 1. Initialize two sentences.
st_1 = "dogs chase cats"
st_2 = "dogs hate cats"

# 2. Create set of words from strings
st_1_wrds = set(st_1.split())
st_2_wrds = set(st_2.split())

unq_wrds = st_1_wrds.union(st_2_wrds)

a = [ 1 if w in st_1_wrds else 0 for w in unq_wrds ]
b = [ 1 if w in st_2_wrds else 0 for w in unq_wrds]

print a
print b
print jaccard_similarity_score(a,b)
```

The output is as follows:

```
[1, 0, 1, 1]
[1, 1, 1, 0]
0.5
```

Writing a list

A list is a container object and sequence type. They are similar to tuples except that they are homogenous and mutable. A list allows append operations. They can also be used as either a stack or queue. Unlike tuples, lists are expandable; you can add elements to a list using the append function after its creation.

Getting ready

Similar to how we saw tuples, we will see lists as fragmented codes where we will concentrate on the creation and manipulation activity instead of having a full program as we did with dictionaries.

How to do it...

Let's look at some Python scripts demonstrating the list creation and manipulation activities:

```
# 1.Let us look at a quick example of list creation.  
a = range(1,10)  
print a  
b = ["a","b","c"]  
print b  
  
# 2.List can be accessed through indexing. Indexing starts at 0.  
print a[0]  
  
# 3.With negative indexing the elements of a list are accessed from  
backwards.  
a[-1]  
  
# 4.Slicing is accessing a subset of list by providing two indices.  
print a[1:3] # prints [2, 3]  
print a[1:] # prints [2, 3, 4, 5, 6, 7, 8, 9]  
print a[-1:] # prints [9]  
print a[::-1] # prints [1, 2, 3, 4, 5, 6, 7, 8]  
  
#5.List concatenation  
a = [1,2]  
b = [3,4]  
print a + b # prints [1, 2, 3, 4]  
  
# 6.      List min max  
print min(a),max(a)  
  
# 7.      in and not in  
if 1 in a:  
    print "Element 1 is available in list a"  
else:  
    print "Element 1 is available in tuple a"  
  
# 8. Appending and extending list
```

```
a = range(1,10)
print a
a.append(10)
print a

# 9.List as a stack
a_stack = []

a_stack.append(1)
a_stack.append(2)
a_stack.append(3)

print a_stack.pop()
print a_stack.pop()
print a_stack.pop()

# 10.List as queue
a_queue = []

a_queue.append(1)
a_queue.append(2)
a_queue.append(3)

print a_queue.pop(0)
print a_queue.pop(0)
print a_queue.pop(0)

# 11.      List sort and reverse
from random import shuffle
a = range(1,20)
shuffle(a)
print a
a.sort()
print a

a.reverse()
print a
```

How it works...

In step 1, we saw different ways of creating a list. Note that we have only homogeneous elements. There can be duplicates unlike a set. Steps 2,3,4,5,6, and 7 are similar to the tuple steps. We will not elaborate on these steps. They cover the indexing, slicing, concatenation, minmax, and in and not in operations similar to a tuple.

Step 8 presents the append and extend operations. This is where a list starts to differ from a tuple. (Of course, we know that these lists are homogeneous.) Let's look at the output of the first part of the code:

```
>>> a = range(1,10)
>>> print a

[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a.append(10)
>>> print a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>
```

We can see that 10 is added to the a list.

The following output is of the second part where extend is shown:

```
>>> b=range(11,15)
>>> a.extend(b)
>>> print a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>>
```

We extended the original a list by another list, b.

In step 9, we will show how a list can be used as a stack. The `pop()` function helps to retrieve the last element appended to the list. The output is as follows:

```
3
2
1
```

The last element to be appended is the first element to be retrieved **Last In, First Out (LIFO)** style as in stacks.

In step 10, we will implement a queue using a list. The `pop()` function with zero as a parameter indicates that the index of the element to be retrieved has been passed. The output is as follows:

```
1
2
3
```

The output adheres to the LIFO style of a queue. However, this is not a very efficient method. Popping the first element is not optimal because of the way a list is implemented. An efficient way to perform this operation is to use the deque data structure explained in the next section.

The final step details the sort and reverse operations in a list. A list has a built-in function, `sort()`, to sort the elements of a list. By default, it sorts in an ascending order. Sorting is explained in detail in a later section of this chapter. The `reverse()` function will reverse the elements of a list.

We will first create a list with elements from 1 to 19:

```
a = range(1,20)
```

We will shuffle the elements using the `shuffle()` function from a module `random`. This shuffles the elements so that we can demonstrate the sort operations. The shuffled output is as follows:

```
[19, 14, 11, 12, 4, 13, 17, 5, 2, 3, 1, 16, 8, 15, 18, 6, 7, 9, 10]
```

Now, `a.sort()` does an in-place sort and when we print `a`, we will get the following output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

The `a.reverse()` is also an in-place operation that produces the following output:

```
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

There's more...

The deque stands for double-ended queue. Unlike stack and queues, which can be appended and popped in only one direction, the append and pop operations can be done at both ends with deque:

<https://docs.python.org/2/library/collections.html#collections.deque>

Creating a list from another list - list comprehension

Comprehension is a way to create a sequence from another sequence. For example, we can create a list from another list or tuple. Let's look at a list comprehension. Typically, a list comprehension involves the following features:

- ▶ A sequence, say a list whose elements we are interested in
- ▶ A variable representing the elements of the sequence
- ▶ An output expression that is responsible for producing the output sequence using the elements of the input sequence
- ▶ An optional predicate expression

Getting ready

Let's define a simple problem in order to understand all the different elements involved in comprehension. With an input list with positive and negative numbers, we need an output list that is the square of all the negative elements.

How to do it...

In the following script, we will show a simple example of list comprehension:

```
# 1.      Let us define a simple list with some positive and negative
numbers.
a = [1,2,-1,-2,3,4,-3,-4]

# 2.      Now let us write our list comprehension.
# pow() a power function takes two input and
# its output is the first variable raised to the power of the second.
b = [pow(x,2) for x in a if x < 0]

# 3.      Finally let us see the output, i.e. the newly created list
b.
print b
```

How it works...

This example is written in a way to explain the various components of comprehension. Let's look at step 2:

```
b = [pow(x,2) for x in a if x < 0]
```

This code is explained as follows:

- ▶ Our input list is `a` and output list is `b`
- ▶ We will use a variable `x` to represent each element in the list
- ▶ The `pow(x, 2)` is the output expression, which uses the elements in the input to produce the output list
- ▶ Finally, `if x < 0` is the predicate expression that controls which elements of the input list are used to produce the output list

There's more...

The comprehension syntax is exactly the same as a dictionary. A simple example will illustrate the following:

```
a = {'a':1,'b':2,'c':3}
b = {x:pow(y,2) for x,y in a.items()}
print b
```

In the preceding example, we created a new dictionary, b from the input dictionary, a. The output is as follows:

```
{'a': 1, 'c': 9, 'b': 4}
```

You can see that we retained the keys of the a dictionary, but now the new values are a square of the original values in a. A point to note is the use of curly braces instead of brackets during the comprehension.

We can do comprehension for tuples with a small trick. See the following example:

```
def process(x):
    if isinstance(x,str):
        return x.lower()
    elif isinstance(x,int):
        return x*x
    else:
        return -9

a = (1,2,-1,-2,'D',3,4,-3,'A')
b = tuple(process(x) for x in a)

print b
```

Instead of the pow() function, we used a new process function. I will leave it to you as an exercise to decipher what the process function does. Note that we followed the same syntax for a comprehension list; however, we used braces instead of brackets. The output of this program is as follows:

```
<generator object <genexpr> at 0x05E87D00>
```

Oops! We wanted a tuple but ended up with a generator (more on generators in the later sections). The right way to do it is as follows:

```
b = tuple(process(x) for x in a)
```

Now, the print b statement will produce the following output:

```
(1, 4, 1, 4, 'd', 9, 16, 9, 'a')
```

Python comprehension is based on the set builder notation:

http://en.wikipedia.org/wiki/Set-builder_notation

Itertools.dropwhile:

<https://docs.python.org/2/library/itertools.html#itertools.dropwhile>

With a predicate and sequence, dropwhile will return only those items in the sequence that satisfies the predicate.

Using iterators

It's a no-brainer that a key input for a data science program is data. Data may vary in size—some of them may fit into memory and some may not. The record access mechanism can vary from one data format to another. Interestingly, different algorithms may demand chunks of varying length to process. For example, let's say that you are writing a stochastic gradient descent algorithm and you want to pass chunks of 5,000 records in each epoch, it will be very nice to have an abstraction that can handle the accessing of the data, understanding the data format, looping through the data, and providing the caller with the required data. This will result in a clean code. Most of the time, the interesting part lies in what we do with the data and not how we access the data. Python provides us with an elegant way in the form of iterators to handle all of these requirements.

Getting ready

An iterator in Python implements an iterator pattern. It allows us to go over a sequence one by one without materializing the whole sequence!

How to do it...

Let's create a simple iterator called simple counter and provide it with some code on how to effectively use the iterator:

```
# 1.      Let us write a simple iterator.
class SimpleCounter(object):
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        'Returns itself as an iterator object'
        return self

    def next(self):
```

```
'Returns the next value till current is lower than end'
if self.current > self.end:

    raise StopIteration
else:
    self.current += 1
    return self.current - 1

# 2.      Now let us try to access the iterator
c = SimpleCounter(1,3)
print c.next()
print c.next()
print c.next()
print c.next()

# 3.      Another way to access
for entry in iter(c):
    print entry
```

How it works...

In step 1, we defined a class by the name of `SimpleCounter`. The `__init__` constructor takes two parameters, `start` and `end`, defining the beginning and end of our sequence. Note the two methods, `__iter__` and `next`. Any object in Python that is meant to be an iterator object should support these two functions. The `__iter__` returns the complete class object as an iterator object. The `next` method returns the next value in the iterator.

As shown in step 2, we can access the successive elements in the iterator using the `next()` function. Python also provides us with a convenient function, `iter()`, which can be used in a loop to access elements sequentially as shown in step 3. The `iter()` uses the `next()` function internally.

A point to note is that an iterator object can be used only once. After running the preceding code, we will try to access the iterator as follows:

```
print next(c)
```

It will throw the `StopIteration` exception. Calling `c.next()` after the sequence has exhausted will result in a `StopIteration` exception:

```
raise StopIteration
StopIteration
>>>
```

The `iter()` function handles this exception and exits the loop once the data has been exhausted.

There's more...

Let's see another example of an iterator. Let's say that we need to access a very large file in our program; however, in our program, we will work through it only one line at a time:

```
f = open(some_file_of_interest)
for l in iter(f):
    print l
f.close()
```

In Python, a file object is an iterator; it supports the `iter()` and `next()` functions. Hence, instead of loading the whole file in memory, we can work with a single line at a time.

An iterator gives you the power to write custom code in order to access your data sources in a manner that your application demands.

The following link provides more information about how iterators can be used in various ways in Python:

Infinite iterators, `count()`, `cycle()` and `repeat()` in `itertools`:

<https://docs.python.org/2/library/itertools.html#itertools.cycle>

Generating an iterator and a generator

We saw what an iterator is in the previous recipe; now in this one, let's see how to generate an iterator.

Getting ready

Generators provide a clean syntax to loop through a sequence of values eliminating the need to have the two functions, `__iter__` and `next()`. We don't have to write a class. A point to note is that both generators and iterables produce an iterator.

How it do it...

Let's have a look the following example; it should be easy to follow if you understood comprehension from the previous section. In this case, we have a generator comprehension. If you recall, we tried doing a tuple comprehension in this way and got a generator object:

```
SimpleCounter = (x**2 for x in range(1,10))

tot = 0
for val in SimpleCounter:
```

```
tot+=val  
  
print tot
```

How it works...

It should be clear that the preceding code snippet will find the sum of the squares of a given range; in this case, the range is 1 to 9. (The range function in Python is right-ended.) Using a generator, we created an iterator called `SimpleCounter` and we use it in a `for` loop in order to access the underlying data sequentially. Note that we have not used the `iter()` function here. Notice how clean the code is. We successfully recreated our old `SimpleCounter` class in a very elegant manner.

There's more...

Let's look at how to use the `yield` statement to create a generator:

```
def my_gen(low,high):  
    for x in range(low,high):  
        yield x**2  
  
tot = 0  
  
for val in my_gen(1,10):  
    tot+=val  
print tot
```

In the preceding example, the `my_gen()` function is a generator; we used the `yield` statement to return the output in a sequence.

In the previous section, we mentioned that both a generator and iterables produce an iterator. Let's validate this by trying to call the generator using the `iter` function:

```
gen = (x**2 for x in range(1,10))  
  
for val in iter(gen):  
    print val
```

Before we move on to iterables in our next recipe, a key point to note with a generator is that once we have gone through the sequence, we are done—no more data.



With a generator object, we can go over the sequence only once.

Using iterables

Iterables are similar to generators except for a key difference, we can go on and on with an iterable, that is, once we have exhausted all the elements in a sequence, we can again start accessing it from the beginning unlike a generator.

They are object-based generators that do not hold any state. Any class with the `iter` method that yields data can be used as a stateless object generator.

Getting ready

Let's try to understand iterables with a simple example. This recipe should be easy to follow if you have understood the previous recipes on generators and iterators.

How to do it...

Let's create a simple iterable called `SimpleIterable` and show some scripts that manipulate it:

```
# 1. Let us define a simple class with __iter__ method.
class SimpleIterable(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __iter__(self):
        for x in range(self.start, self.end):
            yield x**2

# Now let us invoke this class and iterate over its values two times.
c = SimpleIterable(1,10)

# First iteration
tot = 0
for val in iter(c):
    tot+=val

print tot

# Second iteration
tot =0
for val in iter(c):
    tot+=val

print tot
```

How it works..

In step 1, we created a simple class that is our iterable. The init constructor takes two arguments, start and end, similar to our earlier example. We defined a function called iter, which will give us our required sequence. In this given range of numbers, the square of these numbers will be returned.

Next, we have two loops. We iterated through our range of numbers, 1 to 10, in the first loop. When we will run the second for loop, you will notice that it once again iterates through the sequence and doesn't raise any exceptions.

See also

- ▶ *Using Iterators* recipe in Chapter 1, Using Python for Data Science
- ▶ *Generating an Iterator - Generators* recipe in Chapter 1, Using Python for Data Science

Passing a function as a variable

Python supports functional programming in addition to imperative paradigms. In the previous sections, we have seen some functional programming constructs without an explicit explanation. Let's go over them in this section. Functions are first-class citizens in Python. They have attributes and they can be referenced and assigned to a variable.

Getting ready

Let's look at the paradigm of passing a function as a variable in Python in this section.

How to do it...

Let's define a simple function and see how it can be used as a variable:

```
# 1.Let us define a simple function.  
def square_input(x):  
    return x*x  
# We will follow it by assigning that function to a variable  
square_me = square_input  
  
# And finally invoke the variable  
print square_me(5)
```

How it works...

We defined a simple function in step 1; with an input, the function returns the square of the input. We assigned this function to a `square_me` variable. Finally, we were able to invoke the function by calling `square_me` with a valid parameter. This demonstrates how a function can be treated as a variable in Python. This is a very import functional programming construct.

Embedding functions in another function

This recipe will explain yet another functional programming construct; defining a function in another function.

Getting ready

Let's take a simple example of writing a function, which will return the sum of the squares of the given input list.

How to do it...

Let's write a simple function to demonstrate a function in another function:

```
# 1.      Let us define a function of function to find the sum of
squares of the given input
def sum_square(x):
    def square_input(x):
        return x*x
    return sum([square_input(x1) for x1 in x])

# Print the output to check for correctness
print sum_square([2,4,5])
```

How it works...

In step 1, you can see that we defined a `square_input()` function in the `sum_square()` function. The parent function uses it to perform the sum of squares operation. In the next step, we called the function and printed its output.

The output produced was as follows:

```
[4, 9, 16]
```

Passing a function as a parameter

Python supports higher order functions, that is, functions that can accept other functions as arguments.

Getting ready

Let's leverage the `square_input` function defined in the previous example and write a code snippet that will demonstrate how functions can be passed as parameters.

How to do it...

Let's now demonstrate how to pass a function as a parameter:

```
from math import log

def square_input(x):
    return x*x

# 1.      Define a generic function, which will take another function
# as input
# and will apply it on the given input sequence.
def apply_func(func_x,input_x):
    return map(func_x,input_x)

# Let us try to use the apply_func() and verify the results
a = [2,3,4]

print apply_func(square_input,a)
print apply_func(log,a)
```

How it works...

In step 1, we defined a `apply_func` function with two variables. The first variable is a function and second one is a sequence. As you can see, we used the `map` function (more on this function in the recipes to follow) to apply the given function to all the elements of the sequence.

Next, we invoked `apply_func` on a list `a`; first with the `square_input` function followed by a `log` function. The output is as follows:

```
[4,  9, 16]
```

As you can see, the elements of `a` are all squared. The map applies the `square_input` function to all the elements in the sequence:

```
[0.69314718055994529, 1.0986122886681098, 1.3862943611198906]
```

Similarly, `log` is applied on all the elements in the sequence.

Returning a function

In this section, let's look at the functions that will return another function.

Getting ready

Let's take a high school example and try to explain the use of functions returning functions.

Our problem is we are given a cylinder of radius `r` and we would like to know the volume of it for different heights:

```
http://www.mathopenref.com/cylindervolume.html
```

```
Volume = area * height = pi * r^2 * h
```

The preceding formula gives the exact cubic units that will fill a cylinder.

How to do it...

Let's write a simple function to demonstrate the concept of a function returning a function. In addition, we will write a small piece of code to show the usage:

```
# 1.      Let us define a function which will explain our
# concept of function returning a function.
def cylinder_vol(r):
    pi = 3.141
    def get_vol(h):
        return pi * r**2 * h
    return get_vol

# 2.      Let us define a radius and find get a volume function,
# which can now find out the volume for the given radius and any
height.
radius = 10
find_volume = cylinder_vol(radius)

# 3.      Let us try to find out the volume for different heights
height = 10
print "Volume of cylinder of radius %d and height %d = %.2f  cubic
units" \
```

```
% (radius,height,find_volume(height))

height = 20
print "Volume of cylinder of radius %d and height %d = %.2f cubic
units" \
      %(radius,height,find_volume(height))
```

How it works...

In step 1, we defined a function called `cylinder_vol()`; it takes a single parameter, `r`, `radius`. In this function, we defined another function, `get_vol()`. The `get_vol()` function has access to `r` and `pi`, takes the height as an argument. For the given radius, `r`, which was the parameter to `cylinder_vol()`, different heights were passed as a parameter to `get_vol()`.

In step 2, we defined a radius; in this case, as ten and invoke the `cylinder_vol()` function with it. It returns the `get_vol()` function, which we stored in a variable named `find_volume`.

In step 3, we invoked `find_volume` with different heights, 10 and 20. Note that we didn't give the radius.

The output produced is as follows:

```
Volume of cylinder of radius 10 and height 10 = 3141.00 cubic units
Volume of cylinder of radius 10 and height 20 = 6282.00 cubic units
```

There's more...

Functools is a module for higher order functions:

<https://docs.python.org/2/library/functools.html>

Altering the function behavior with decorators

Decorators wrap a function and alter their behavior. They are best understood with some working examples. Let's see some decorators in action in this recipe.

Getting ready

Do you recall the section where we explained a function as an argument to another function, function as a variable, and function returning a function? Most important of all, do you remember the cylinder example? If you followed it, decorators should be a piece of cake. In this exercise, we will do a pipeline of cleaning activity on a given string. With a string with mixed casing and punctuation, we will use decorators to write a cleaning routine, which can be extended very easily.

How to do it...

Let's write a simple decorator for the text manipulation:

```
from string import punctuation

def pipeline_wrapper(func):

    def to_lower(x):
        return x.lower()

    def remove_punc(x):
        for p in punctuation:
            x = x.replace(p, '')
        return x

    def wrapper(*args, **kwargs):
        x = to_lower(*args, **kwargs)
        x = remove_punc(x)
        return func(x)
    return wrapper

@pipeline_wrapper
def tokenize_whitespace(inText):
    return inText.split()

s = "string. With. Punctuation?"
print tokenize_whitespace(s)
```

How it works...

Let's start from the last two lines:

```
s = "string. With. Punctuation?"
print tokenize_whitespace(s)
```

We declared a string variable. We want to clean the string. In our case, we want the following features:

- ▶ We want the string to be in lowercase
- ▶ We want to strip the punctuation
- ▶ We want to return a list of words

You can see that we called the `tokenize_whitespace` function with the string, `s`, as a parameter. Let's look at the `tokenize_whitespace` function:

```
@pipeline_wrapper
def tokenize_whitespace(inText):
    return inText.split()
```

We see that this is very simple function and with a string input, the function splits it by a space and returns a list of words. We will alter the behavior of this function using decorators. You can see that the decorator that we will use for this function is `@pipeline_wrapper`. This is an easier way of calling the following:

```
tokenize_whitespace = pipeline_wrapper (clean_tokens)
```

Now, let's look at the decorator function:

```
def pipeline_wrapper(func):

    def to_lower(x):
        return x.lower()
    def remove_punc(x):
        for p in punctuation:
            x = x.replace(p, '')
        return x
    def wrapper(*args,**kwargs):
        x = to_lower(*args,**kwargs)
        x = remove_punc(x)
        return func(x)
    return wrapper
```

You can see that `pipeline_wrapper` returns the `wrapper` function. In the `wrapper` function, you can see that the final return statement returns `func`; this is the original function passed by us to the `wrapper`. The `wrapper` modifies the behavior of our original `tokenize_whitespace` function. The input to `tokenize_whitespace` is modified first by the `to_lower()` function, which changes the input string to lowercase, followed by the `remove_punc()` function, which removes the punctuation. The final output is as follows:

```
['string', 'with', 'punctuation']
```

Exactly what we wanted—the punctuation stripped, strings converted to lowercase, and finally, a list of words.

Creating anonymous functions with lambda

Anonymous functions are created using the `lambda` statement in Python. Functions that are not bound to a name are called anonymous functions.

Getting ready

If you followed the section on passing functions as a parameter, the example in this section is very similar to it. We passed a predefined function in that section; here we will pass a `lambda` function.

How to do it...

We will see a simple example with a toy dataset to explain anonymous functions in Python:

```
# 1.      Create a simple list and a function similar to the
# one in functions as parameter section.

a =[10,20,30]

def do_list(a_list,func):
    total = 0
    for element in a_list:
        total+=func(element)
    return total

print do_list(a,lambda x:x**2)
print do_list(a,lambda x:x**3)

b =[lambda x: x%3 ==0  for x in a ]
```

How it works...

In step 1, we have a function called `do_list` that accepts another function as an argument. With a list and function, `do_list` applies the input function over the elements of the given list, sums up the transformed values, and returns the results.

Next, we will invoke the `do-list` function, the first parameter is our input list `a`, and the second parameter is our `lambda` function. Let's decode our `lambda` function:

```
lambda x:x**2
```

An anonymous function is declared using the keyword, lambda; it's followed by defining a parameter for the function. In this case, x is the name of the parameter passed to this anonymous function. The expression succeeding the colon operator is the return value. The input parameter is evaluated using the expression and returned as the output. In this input, the square of the input is returned as the output. In the next print statement, we have a lambda function, which returns the cube of the given input.

Using the map function

Map is a built-in Python function. It takes a function and an iterable for an argument:

```
map(aFunction, iterable)
```

The function is applied on all the elements of the iterable and results are returned as a list. As a function is passed to map, lambda is most commonly used along with map.

Getting ready

Let's look at a very simple example using the map function.

How to do it...

Let's see an example on how to use a map function:

```
#First let us declare a list.  
a = [10,20,30]  
# Let us now call the map function in our Print statement.  
print map(lambda x:x**2,a)
```

How it works...

This is very similar to the code in the previous recipe. A map functions takes two parameters. The first one is a function and second one is a sequence. In our example code, we used an anonymous function:

```
lambda x:x**2
```

This function squares the given input. We also passed a list to map.

Map applies a function that squares all the elements in the given list and returns the result as a list.

The output is as follows:

```
[100,400,900]
```

There's more...

Similarly, any other function can be applied to a list:

```
print map(lambda x:x**3,a)
```

Using map, we can replace the code snippet in the previous recipe with a single line:

```
print sum(map(lambda x:x**2,a))
print sum(map(lambda x:x**3,a))
```

Map expects an N-argument function if we have N-sequences. Let's see an example to understand this:

```
a = [10,20,30]
b = [1,2,3]

print map(pow,a,b)
```

We passed two sequences a and b to our map function. Notice that the function passed is the power function. It takes two arguments. Let's see the result of the preceding code snippet:

```
[10, 400, 27000]
>>>
```

As you can see, the elements of list a is raised to the power of value in the same position in list b. A point to note is that both the lists should be of the same size; if not, Python will fill the smaller list with None. Though our examples are operating on a list, any iterable can be passed to a map function.

Working with filters

True to its name, filter filters elements from a sequence based on the given function. With a sequence of negative and positive numbers, we can use a filter function to, say, filter out all the negative numbers. Filter is a built-in Python function. It takes a function and an iterable for an argument:

```
Filter(aFunction, iterable)
```

The function that is passed as an argument is returned as a Boolean value based on a test.

The function is applied on all the elements of the iterable and all the items that are returned as true when the function is applied over them are returned as a list. An anonymous function, lambda, is most commonly used along with filter.

Getting ready

Let's look at a simple code to see the filter function in action.

How to do it...

Let's see an example on how to use a filter function:

```
# Let us declare a list.  
a = [10,20,30,40,50]  
# Let us apply Filter function on all the elements of the list.  
print filter(lambda x:x>10,a)
```

How it works...

The lambda function that we use here is very simple; it returns true if the given value is greater than ten, false otherwise. Our print statement gives the following result:

```
[20, 30, 40, 50]
```

As you can see, only elements greater than ten are returned.

Using zip and izip

Zip takes two equal length collections and merges them together in pairs. Zip is a built-in Python function.

Getting ready

Let's demonstrate zip using a very simple example.

How to do it...

Let's pass two sequences to a zip function and print the output:

```
print zip(range(1,5),range(1,5))
```

How it works...

The two parameters to our zip function are two lists, both with values ranging from 1 to 5.

A range function takes three parameters. The starting value of the list, ending value of the list, and a step value. The default step value is one. In our case, we passed 1 and 5 as the starting and ending values of the list. Remember that Python is right-closed, so the range (1, 5) will return a list as follows:

```
[1, 2, 3, 4]
```

We pass the two sequences to the zip function and the result is as follows:

```
[(1, 1), (2, 2), (3, 3), (4, 4)]
```

Keep in mind that both the collections should be of the same size; if not, then the output is truncated to the size of the shortest collection.

There's more...

Now, look at the following code:

```
x,y = zip(*out)
print x,y
```

Can you guess what the output is?

Let's see what the * operator does. A * operator unpacks a collection in their positional arguments:

```
a = (2,3)
print pow(*a)
```

The power operation takes two arguments. Now a is a tuple; as you can see, the * operator splits the tuple into two separate arguments. The * operator unpacks the tuple in 2 and 3. They are passed as parameters, pow(2, 3), and we get the output, 8.

The ** operator can be used to unpack a dictionary. Look at the following snippet:

```
a_dict = {"x":10,"y":10,"z":10,"x1":10,"y1":10,"z1":10}
```

The ** operator unpacks a dictionary as a set of named arguments. In this case, we will get an output, 6, when we apply the ** operator to a dictionary. Look at the following function, which takes six arguments:

```
def dist(x,y,z,x1,y1,z1):
    return abs((x-x1)+(y-y1)+(z-z1))

print dist(**a_dict)
```

The output of the print statement is zero.

Armed with these two operators, we can write a function without any restrictions on the number of variables that it can ingest:

```
def any_sum(*args):
    tot = 0
    for arg in args:
        tot+=arg
    return tot

print any_sum(1,2)
print any_sum(1,2,3)
```

As you can see, in the preceding code snippet, the `any_sum` function can now work on any number of variables. A curious reader may comment about why not use a list instead as an argument to the `any_sum` function, where we can now pass a list of values. Very well, yes in this case, but we will soon encounter cases where we really don't know what kind of arguments will be passed.

Back to the zip utility. One drawback with `zip` is that it can compute the list all at once. This may be an issue when we have two very large lists. The `izip` comes to the rescue in these cases. They compute the elements only when requested. The `izip` is a part of `itertools`. Please refer to the `itertools` recipe for more details.

See also

- ▶ *Working with Itertools recipe in Chapter 1, Using Python for Data Science*

Processing arrays from the tabular data

The meat of any data science application is to find an appropriate data handling routine for a given problem. In the case of machine learning, it's either the supervised or unsupervised method to predict or classify the data. Even before this step, a good amount of time is spent in the data transformation and making the data suitable for these methods.

Usually, data is made available to a data science program in many ways. A data science programmer is faced with the challenge of accessing the data and making it available to a later part of his code using the Python data structure. Mastering ways to access data through Python will be very handy when writing a data science program as it will allow you to jump to the meat of the problem very quickly.

Typically, data is available as a text file, separated by either a comma or tab. A Python built-in file object utility can be used in this case. As we saw earlier, a file object implements the `__iter__()` and `next()` methods. This allows us to work on very large files, which do not fit into memory, by reading only a small chunk of the files at a time.

Python machine learning libraries such as scikit-learn works on the NumPy libraries. In this section, we will see ways of efficiently reading external data and converting it to NumPy arrays for the downstream data processing.

Getting ready

NumPy provides us with a function called `genfromtext` to create NumPy arrays from tabular data. Once the data is available as NumPy arrays, it's much easier for the downstream systems to process this data. Let's look at how we can leverage `genfromtext`. The following code was written using the NumPy version 1.8.0.

How to do it...

Let's import the necessary libraries to start with. We will proceed to define a sample input. Finally, we will demonstrate how to process tabular data.

```
# 1.      Let us simulate a small tablular input using StringIO
import numpy as np
from StringIO import StringIO
in_data = StringIO("10,20,30\n56,89,90\n33,46,89")

# 2.Read the input using numpy's genfromtext to create a nummpy
array.
data = np.genfromtxt(in_data,dtype=int,delimiter=", ")

# cases where we may not need to use some columns.
in_data = StringIO("10,20,30\n56,89,90\n33,46,89")
data = np.genfromtxt(in_data,dtype=int,delimiter=",",usecols=(0,1))

# providing column names
in_data = StringIO("10,20,30\n56,89,90\n33,46,89")
data = np.genfromtxt(in_data,dtype=int,delimiter=",",names="a,b,c")

# using column names from data
in_data = StringIO("a,b,c\n10,20,30\n56,89,90\n33,46,89")
data = np.genfromtxt(in_data,dtype=int,delimiter=",",names=True)
```

How it works...

In step 1, we simulated a tabulated data using the `StringIO` utility. We have three rows and three columns. The rows are new line-delimited and columns are comma-delimited.

In step 2, we used `genfromtxt` from NumPy to ingest the data as a NumPy array.

The first argument to `genfromtxt` is the source of the file and filename; in our case, it's the `StringIO` object. The input is comma-delimited; the delimiter argument allows us to specify the same. After running the preceding code, the data value is as follows:

```
>>> data
array([[10, 20, 30],
       [56, 89, 90],
       [33, 46, 89]])
```

As you can see, we successfully loaded the data from the string in a NumPy array.

There's more...

Various parameters and default values of the same are shown here for the `genfromtxt` function:

```
genfromtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None,
skiprows=0, skip_header=0, skip_footer=0, converters=None,
missing='', missing_values=None, filling_values=None, usecols=None,
names=None, excludelist=None, deletechars=None, replace_space='_',
autostrip=False, case_sensitive=True, defaultfmt='f%i', unpack=None,
usemask=False, loose=True, invalid_raise=True)
```

The only mandatory argument is the name of the source of the data. In our case, we used a `StringIO` object. It can be a string corresponding to the name of a file or an object similar to a file with a `read` method. It can also be a URL of a remote file.

The first step is to split the given line into columns. Once the file is open to be read, `genfromtxt` splits the non-empty lines into a sequence of strings. Empty lines are ignored and so are the commented lines. The `comments` option helps `gentext` decide which are the comment lines. The strings are split into columns based on a delimiter specified by the `delimiter` option. In our example case, we used a `,` delimiter. A `/t` is also a very popular delimiter. By default, the delimiter is `None` in `gentext`, which means that it assumes that the line is split into columns through whitespaces.

Typically, when lines are changed to a sequence of strings and subsequently the columns are extracted, the individual columns are not stripped of the leading or trailing whitespaces. In a later part of the code, this needs to be handled, especially if some of the variables are used as keys in a dictionary. For example, if the leading or trailing whitespaces are not handled consistently, this may lead to a bug/error in the code. Setting `autostrip=True` helps avoid this problem.

Many times, we want to skip, say, top `n` rows or bottom `n` rows while reading a file. This may be due to the presence of headers or footers. The `skip_header = n` will skip the first `n` lines while reading and similarly, `skip_footer = n` will skip the last `n` lines.

Similar to unwanted rows, we may encounter many cases where we may not need to use some columns. The `usecols` argument is used to specify the list of columns that we are interested in:

```
in_data = StringIO("10,20,30\n56,89,90\n33,46,89")  
  
data = np.genfromtxt(in_data,dtype=int,delimiter=",",usecols=(0,1))
```

As you can see in the preceding example, we selected only two columns, column 0 and 1. The data object looks as follows:

```
>>> data  
array([[10, 20],  
       [56, 89],  
       [33, 46]])
```

Custom column names can be provided using the `names` argument. A string argument with comma-separated column names looks as follows:

```
in_data = StringIO("10,20,30\n56,89,90\n33,46,89")  
data = np.genfromtxt(in_data,dtype=int,delimiter=",",names="a,b,c")  
  
>>> data  
array([(10, 20, 30), (56, 89, 90), (33, 46, 89)],  
      dtype=[('a', '<i4'), ('b', '<i4'), ('c', '<i4')])
```

By having `names` to true, the first row in the input data is used as a column header:

```
in_data = StringIO("a,b,c\n10,20,30\n56,89,90\n33,46,89")  
data = np.genfromtxt(in_data,dtype=int,delimiter=",",names=True)  
  
>>> data  
array([(10, 20, 30), (56, 89, 90), (33, 46, 89)],  
      dtype=[('a', '<i4'), ('b', '<i4'), ('c', '<i4')])
```

Another simple method from NumPy to create NumPy arrays from the text input is `loadtxt`:

<http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html>

This is less sophisticated than `genfromtxt`; if you need a simple reader without any sophisticated data handling mechanisms such as handling missing values, you can opt for `loadtxt`.

However, if we are not interested in loading the data as a NumPy array but want to load it as a list, Python provides us with a default csv library:

<https://docs.python.org/2/library/csv.html>

An interesting method in the preceding csv library is `csv.Sniffer.sniff()`. If we have a very large csv file and we want to understand its structure, we can use `sniff()`. This will return a dialect subclass, which has most of the properties of the csv file.

Preprocessing the columns

Often the data that we get is not in the format we can consume. A lot of data processing called data preprocessing steps in machine learning terminology has to be applied. One way to work through this hurdle is to ingest all the input as strings and carry on with the required data transformation at the later stages. Another way is to perform these changes at the source. The `genfromtxt` provides us with some functionalities in order to perform this data transformation while reading from the source.

Getting ready

Consider the following lines of text:

```
30kg,inr2000,31.11,56.33,1  
52kg,inr8000.35,12,16.7,2
```

This is a typical example of how we get data in real life. The first two columns have a string kg and inr attached to the rear and front of the actual values.

Let's try to ingest this data in a NumPy array as follows:

```
in_data = StringIO("30kg,inr2000,31.11,56.33,1\  
n52kg,inr8000.35,12,16.7,2")  
data = np.genfromtxt(in_data,delimiter=",")
```

This results in the following:

```
>>> data  
array([[    nan,      nan,  31.11,   56.33,    1.  ],  
       [    nan,      nan,  12.  ,   16.7 ,    2.  ]])
```

As you can see, the first two columns are not read.

How to do it...

Let's import the necessary libraries to start with. We will proceed to define a sample input. Finally, we will demonstrate data preprocessing.

```
import numpy as np  
from StringIO import StringIO  
  
# Define a data set
```

```
in_data = StringIO("30kg,inr2000,31.11,56.33,1\  
n52kg,inr8000.35,12,16.7,2")  
  
# 1.Let us define two data pre-processing using lambda functions,  
strip_func_1 = lambda x : float(x.rstrip("kg"))  
strip_func_2 = lambda x : float(x.lstrip("inr"))  
  
# 2.Let us now create a dictionary of these functions,  
convert_funcs = {0:strip_func_1,1:strip_func_2}  
  
# 3.Now provide this dictionary of functions to genfromtxt.  
data = np.genfromtxt(in_data,delimiter=",", converters=convert_funcs)  
  
# Using a lambda function to handle conversions  
in_data = StringIO("10,20,30\n56,,90\n33,46,89")  
mss_func = lambda x : float(x.strip() or -999)  
data = np.genfromtxt(in_data,delimiter=",", converters={1:mss_func})
```

How it works...

In step 1, we defined two lambda functions, one for column 1 where we need to strip the string 'kg' from the right-hand side and another to strip the string 'inr' from the left-hand side of column 2.

In step 2, we will go ahead and define a dictionary where the key is the column name to which the function has to be applied and the value is the function. This dictionary is passed as a parameter with the `converters` name in `genfromtext`.

Now the output is as follows:

```
>>> data  
array([[ 3.0000000e+01,   2.0000000e+03,   3.1110000e+01,  
        5.6330000e+01,   1.0000000e+00],  
       [ 5.2000000e+01,   8.0003500e+03,   1.2000000e+01,  
        1.6700000e+01,   2.0000000e+00]])
```

Note that Nan has vanished, giving way to an actual value from the input.

There's more...

Converters can also be used to handle the missing values in an input record through a lambda function:

```
in_data = StringIO("10,20,30\n56,,90\n33,46,89")  
mss_func = lambda x : float(x.strip() or -999)  
data = np.genfromtxt(in_data,delimiter=",", converters={1:mss_func})
```

The lambda function returns -999 for the missing values. In our input, the second column of the second row is empty and this should be replaced by -999. The final output looks as follows:

```
>>> data
array([[ 10.,   20.,   30.],
       [ 56., -999.,   90.],
       [ 33.,   46.,   89.]])
```

Refer to the SciPy documentations given here for more details:

<http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html>

<http://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html>

Sorting lists

We will start with sorting a list and then move on to sorting other iterables.

Getting ready

There are two ways to proceed with the sorting. The first way is to use the built-in sort function in the list and the other way is to use the sorted function. Let's work it out through an example.

How to do it...

Let's see how to leverage the sort and sorted functions:

```
# Let us look at a very small code snippet, which does sorting of a
# given list.
a = [8, 0, 3, 4, 5, 2, 9, 6, 7, 1]
b = [8, 0, 3, 4, 5, 2, 9, 6, 7, 1]

print a
a.sort()
print a

print b
b_s = sorted(b)
print b_s
```

How it works...

We declared two lists, `a` and `b`, with the same elements. As a convenience way to verify the output, we will print list `a`:

```
[8, 0, 3, 4, 5, 2, 9, 6, 7, 1]
```

We used the `sort` function available with the list data type, `a.sort()`, to perform an in-place sort. The following print statement shows that the list has now been sorted:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Now, we will use the `sorted` function. This function performs the sorting on the list and returns a new sorted list. You can see that we invoked it as `sorted(b)` and stored the output in `b_s`. The print statement against `b_s` yields a sorted output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

There's more...

The `sort` function is only available for a list data type. By default, sorting is done in an ascending order; this can be controlled by a `reverse` parameter to the `sort` function. By default, `reverse` is set to `False`:

```
>>> a = [8, 0, 3, 4, 5, 2, 9, 6, 7, 1]
>>> print a
[8, 0, 3, 4, 5, 2, 9, 6, 7, 1]
>>> a.sort(reverse=True)
>>> print a
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>>
Now, we have a descending order sorting.
For other iterables, we have to fall back on the sorted function.
Let's look at a tuple example:
>>> a = (8, 0, 3, 4, 5, 2, 9, 6, 7, 1)
>>> sorted(a)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Sorting with a key

Till now, we saw all the examples where a list or sequence was sorted by elements. Let's now proceed to see if we can sort it using keys. In the previous example, elements were the keys. In the real world, there are more complicated records where a record contains multiple columns and we would like to sort with one or more columns). We will work out our examples through a list of tuples and the same applies to the other sequence objects.

Getting ready

In our example, a single tuple represents a person's record, which includes his name, ID, and age. Let's write a sorting in order to sort by the various fields.

How to do it...

Let's define a record-like structure using a list and tuples. We will use this data to demonstrate the data sorting with a key:

```
#1.The first step is to create a list of tuples, which we will use to test our sorting.
```

```
employee_records = [ ('joe',1,53),('beck',2,26), \
                     ('ele',6,32),('neo',3,45), \
                     ('christ',5,33),('trinity',4,29), \
                     ]
```



```
# 2.Let us now sort it by employee name
print sorted(employee_records,key=lambda emp : emp[0])
"""
It prints as follows
[('beck', 2, 26), ('christ', 5, 33), ('ele', 6, 32), ('joe', 1, 53),
 ('neo', 3, 45), ('trinity', 4, 29)]
"""

# 3.Let us now sort it by employee id
print sorted(employee_records,key=lambda emp : emp[1])
"""
It prints as follows
[('joe', 1, 53), ('beck', 2, 26), ('neo', 3, 45), ('trinity', 4, 29),
 ('christ', 5, 33), ('ele', 6, 32)]
"""

# 4.Finally we sort it with employee age
print sorted(employee_records,key=lambda emp : emp[2])
"""
Its prints as follows
[('beck', 2, 26), ('trinity', 4, 29), ('ele', 6, 32), ('christ', 5,
 33), ('neo', 3, 45), ('joe', 1, 53)]
"""


```

How it works...

In our example, each record has three fields: name, identification, and age. We used the lambda function to pass a key by which we need to sort the given records. In step 2, we passed the name as the key to sort. Similarly, in steps 2 and 3, we passed the ID and age as the keys. We can see the outputs in various steps; the outputs are sorted by the particular key that we want them to.

There's more...

Due to the importance of sorting by a key, Python provides a convenient function to access keys instead of writing lambdas. The operator module has the `itemgetter`, `attrgetter`, and `methodcaller` functions. The sorting example that we saw can be written as follows using `itemgetter`:

```
from operator import itemgetter
employee_records = [ ('joe',1,53),('beck',2,26), \
                     ('ele',6,32),('neo',3,45), \
                     ('christ',5,33),('trinity',4,29), \
                     ]
print sorted(employee_records,key=itemgetter(0))
"""
[('beck', 2, 26), ('christ', 5, 33), ('ele', 6, 32), ('joe', 1, 53),
 ('neo', 3, 45), ('trinity', 4, 29)]
"""

print sorted(employee_records,key=itemgetter(1))
"""
[('joe', 1, 53), ('beck', 2, 26), ('neo', 3, 45), ('trinity', 4, 29),
 ('christ', 5, 33), ('ele', 6, 32)]
"""

print sorted(employee_records,key=itemgetter(2))
"""
[('beck', 2, 26), ('trinity', 4, 29), ('ele', 6, 32), ('christ', 5,
 33), ('neo', 3, 45), ('joe', 1, 53)]
"""
```

Note that we have not used the lambda functions, rather used `itemgetter` to specify the key by which we need to sort. More than one field can be given as an input to `itemgetter` when we need multiple level sorting; for example, let's say that we need to sort by name and then by age, our code would be as follows:

```
>>> sorted(employee_records,key=itemgetter(0,1))
[('beck', 2, 26), ('christ', 5, 33), ('ele', 6, 32), ('joe', 1, 53),
 ('neo', 3, 45), ('trinity', 4, 29)]
```

The `attrgetter` and `methodcaller` comes in handy when the elements of our iterable are class objects. Look at the following example:

```
# Let us now enclose the employee records as class objects,
class employee(object):
    def __init__(self, name, id, age):
        self.name = name
        self.id = id
        self.age = age
    def pretty_print(self):
        print self.name, self.id, self.age

# Now let us populate a list with these class objects.
employee_records = []
emp1 = employee('joe', 1, 53)
emp2 = employee('beck', 2, 26)
emp3 = employee('ele', 6, 32)

employee_records.append(emp1)
employee_records.append(emp2)
employee_records.append(emp3)

# Print the records
for emp in employee_records:
    emp.pretty_print()

from operator import attrgetter
employee_records_sorted = sorted(employee_
records, key=attrgetter('age'))
# Now let us print the sorted list,
for emp in employee_records_sorted:
    emp.pretty_print()
```

The constructor initializes the class with three variables: name, age, and ID. We also have the `pretty_print` method to print the values of the class object.

Next, let's populate a list with these class objects:

```
employee_records = []
emp1 = employee('joe', 1, 53)
emp2 = employee('beck', 2, 26)
emp3 = employee('ele', 6, 32)
```

```
employee_records.append(emp1)
employee_records.append(emp2)
employee_records.append(emp3)
```

Now, we have a list of employee objects. There are three variables in each object: name, ID, and age. Let's print the list to see the order:

```
joe 1 53
beck 2 26
ele 6 32
```

As you can see, the order of the insertion has been preserved. Now, let's use `attrgetter` to sort the list with the age field:

```
employee_records_sorted = sorted(employee_
records, key=attrgetter('age'))
```

Let's print the sorted list.

The output is as follows:

```
beck 2 26
ele 6 32
joe 1 53
```

You can see that the records are now sorted by age.

The `methodcaller` can be used to sort when we want to use a method in our class to decide the sorting. For demonstration purposes, let's add a random method, which divides the age by the ID:

```
class employee(object):
    def __init__(self, name, id, age):
        self.name = name
        self.id = id
        self.age = age

    def pretty_print(self):
        print self.name, self.id, self.age

    def random_method(self):
        return self.age / self.id

# Populate data
employee_records = []
emp1 = employee('joe', 1, 53)
emp2 = employee('beck', 2, 26)
```

```
emp3 = employee('ele',6,32)

employee_records.append(emp1)
employee_records.append(emp2)
employee_records.append(emp3)

from operator import methodcaller
employee_records_sorted = sorted(employee_records, key=methodcaller('random_method'))
for emp in employee_records_sorted:
    emp.pretty_print()
```

We can now sort the list by calling this method:

```
sorted(employee_records, key=methodcaller('random_method'))
```

Let's now print the list in a sorted order and see the output:

```
ele 6 32
beck 2 26
joe 1 53
```

Working with itertools

Itertools includes functions to work with iterables; it is inspired by a functional programming language such as Haskell. They promise to be memory-efficient and very fast.

Getting ready

There are a lot of functions available in Itertools; we will go through some of them as we work it out through examples. A link to the full list of functions has been provided.

How to do it...

Let's proceed to see a set of Python scripts used to demonstrate the usage of itertools:

```
# Load libraries
from itertools import chain,compress,combinations,count,izip,islice

# 1.Chain example, where different iterables can be combined together.
a = [1,2,3]
b = ['a','b','c']
print list(chain(a,b)) # prints [1, 2, 3, 'a', 'b', 'c']
```

```
# 2.Compress example, a data selector, where the data in the first
# iterator
#   is selected based on the second iterator.
a = [1,2,3]
b = [1,0,1]
print list(compress(a,b)) # prints [1, 3]

# 3.From a given list, return n length sub sequences.
a = [1,2,3,4]
print list(combinations(a,2)) # prints [(1, 2), (1, 3), (1, 4), (2,
3), (2, 4), (3, 4)]

# 4.A counter which produces infinite consequent integers, given a
start integer,
a = range(5)
b = izip(count(1),a)
for element in b:
    print element

# 5.      Extract an iterator from another iterator,
# let us say we want an iterator which only returns every
# alternate elements from the input iterator
a = range(100)
b = islice(a,0,100,2)
print list(b)
```

How it works...

Step 1 is pretty straightforward, where two iterables are combined using `chain()`. A point to note is that `chain()` is not realized till it's actually called. Check the following command line:

```
>>> chain(a,b)
<itertools.chain object at 0x060DD0D0>
```

Calling `chain(a,b)` returns the `chain` object. However, when we run the following command, the actual output is produced:

```
>>> list(chain(a,b))
[1, 2, 3, 'a', 'b', 'c']
```

Step 2 describes `compress`. In this example, elements of `a` are selected based on elements in `b`. You can see that in `b`, the second value is zero and hence, the second value in `a` is also not selected.

Step 3 does simple mathematical combinations. We have an input list, `a`, and want the elements of `a` in combinations of two.

Step 4 explains a counter object, which can serve as an infinite resource of a sequence number given a start number. Running the code, we will get the following output:

```
(1, 0)
(2, 1)
(3, 2)
(4, 3)
(5, 4)
```

You can see that we used `izip` here. (`Zip` and `izip` have been covered in previous sections.) Our output is a tuple where the first element is provided by counter and second element is provided by our input list, `a`.

Step 5 details the `islice` operation; `islice` is the same as `slice`, which we covered in the previous section, except that `islice` is memory-efficient and does not realize the complete output unless called upon.

Refer to <https://docs.python.org/2/library/itertools.html> for a complete list of the `itertools`.

2

Python Environments

In this chapter, we will cover the following recipes:

- ▶ Using NumPy libraries
- ▶ Plotting with matplotlib
- ▶ Machine learning with scikit-learn

Introduction

In this chapter, we will introduce you to the Python environment, which will be used extensively throughout this book. We will start with NumPy, which is a Python library that is used to handle arrays and matrices efficiently. It forms the basis for most of the other libraries used in this book. We will then introduce a Python plotting library called matplotlib. Our final recipe is about a machine learning library called scikit-learn.

Using NumPy libraries

NumPy provides an efficient way of handling very large arrays in Python. Most of the Python scientific libraries use NumPy internally for the array and matrix operations. In this book, we will be using NumPy extensively. We will introduce NumPy in this recipe.

Getting ready

We will write a series of Python statements manipulating arrays and matrices, and learn how to use NumPy on the way. Our intent is to get you used to working with NumPy arrays, as NumPy will serve as the basis for most of the recipes in this book.

How to do it...

Let's start by creating some simple matrices and arrays:

```
#Recipe_1a.py
# Importing numpy as np
import numpy as np
# Creating arrays
a_list = [1,2,3]
an_array = np.array(a_list)
# Specify the datatype
an_array = np.array(a_list,dtype=float)

# Creating matrices
a_listoflist = [[1,2,3],[5,6,7],[8,9,10]]
a_matrix = np.matrix(a_listoflist,dtype=float)
```

Now we will write a small convenience function in order to inspect our NumPy objects:

```
#Recipe_1b.py
# A simple function to examine given numpy object
def display_shape(a):
    print
    print a
    print
    print "Nuber of elements in a = %d"%(a.size)
    print "Number of dimensions in a = %d"%(a.ndim)
    print "Rows and Columns in a ",a.shape
    print

display_shape(a_matrix)
```

Let's see some alternate ways of creating arrays:

```
#Recipe_1c.py
# Alternate ways of creating arrays
# 1. Leverage np.arange to create numpy array
created_array = np.arange(1,10,dtype=float)
display_shape(created_array)

# 2. Using np.linspace to create numpy array
created_array = np.linspace(1,10)
display_shape(created_array)

# 3. Create numpy arrays in using np.logspace
created_array = np.logspace(1,10,base=10.0)
display_shape(created_array)
```

```
# Specify step size in arange while creating
# an array. This is where it is different
# from np.linspace
created_array = np.arange(1,10,2,dtype=int)
display_shape(created_array)
```

We will now look at the creation of some special matrices:

```
#Recipe_1d.py
# Create a matrix with all elements as 1
ones_matrix = np.ones((3,3))
display_shape(ones_matrix)
# Create a matrix with all elements as 0
zeros_matrix = np.zeros((3,3))
display_shape(zeros_matrix)

# Identity matrix
# k parameter controls the index of 1
# if k =0, (0,0),(1,1),(2,2) cell values
# are set to 1 in a 3 x 3 matrix
identity_matrix = np.eye(N=3,M=3,k=0)
display_shape(identity_matrix)
identity_matrix = np.eye(N=3,k=1)
display_shape(identity_matrix)
```

Armed with the knowledge of array and matrix creation, let's see some shaping operations:

```
Recipe_1e.py
# Array shaping
a_matrix = np.arange(9).reshape(3,3)
display_shape(a_matrix)
.
.
.
display_shape(back_array)
```

Now, proceed to see some matrix operations:

```
#Recipe_1f.py
# Matrix operations
a_matrix = np.arange(9).reshape(3,3)
b_matrix = np.arange(9).reshape(3,3)
.
.
.
print "f_matrix, row sum", f_matrix.sum(axis=1)
```

Finally, let's see some reverse, copy, and grid operations:

```
#Recipe_1g.py
# reversing elements
display_shape(f_matrix[::-1])
.
.
.
zz = zz.flatten()
```

Let's look at some random number generation routines in the NumPy library:

```
#Recipe_1h.py
# Random numbers
general_random_numbers = np.random.randint(1,100, size=10)
print general_random_numbers
.
.
.
uniform_rnd_numbers = np.random.normal(loc=0.2,scale=0.2,size=(3,3))
```

How it works...

Let's start by including the NumPy library:

```
# Importing numpy as np
import numpy as np
```

Let's proceed with looking at the various ways in which we can create an array in NumPy:

```
# Arrays
a_list = [1,2,3]
an_array = np.array(a_list)
# Specify the datatype
an_array = np.array(a_list,dtype=float)
```

An array can be created from a list. In the preceding example, we declared a list of three elements. We can then use `np.array()` to convert the list to a NumPy one-dimensional array.

The datatype can also be specified, as seen in the last line of the preceding code:

We will now move from arrays to matrices:

```
# Matrices
a_listoflist = [[1,2,3],[5,6,7],[8,9,10]]
a_matrix = np.matrix(a_listoflist,dtype=float)
```

We will create a matrix from a `listoflist`. Once again, we can specify the datatype.

Before we move further, we will define a `display_shape` function. We will use this function frequently further on:

```
def display_shape(a):  
    print  
    print a  
    print  
    print "Nuber of elements in a = %d"%(a.size)  
    print "Number of dimensions in a = %d"%(a.ndim)  
    print "Rows and Columns in a ",a.shape  
    print
```

Every NumPy object has the following three properties:

size: The number of elements in the given NumPy object

ndim: The number of dimensions

shape: The shape returns a tuple with the dimensions of the object

This function prints out all the three properties in addition to printing the original element.

Let's call this function with the matrix that we created previously:

```
display_shape(a_matrix)
```

```
[[ 1.  2.  3.]  
 [ 5.  6.  7.]  
 [ 8.  9.  10.]]  
  
Nuber of elements in a = 9  
Number of dimensions in a = 2  
Rows and Columns in a  (3, 3)
```

As you can see, our matrix has nine elements in it, and there are two dimensions. Finally, we can see the shape displays both the dimensions and number of elements in each dimension. In this case, we have a matrix with three rows and three columns.

Let's now see a couple of other ways of creating arrays:

```
created_array = np.arange(1,10,dtype=float)  
display_shape(created_array)
```

The NumPy `arrange` function returns evenly spaced values in the given interval. In this case, we want an evenly spaced number between 1 and 10. Refer to the following link for more information about `arrange`:

<http://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html>

```
# An alternate way to create array  
created_array = np.linspace(1,10)  
display_shape(created_array)
```

NumPy's `linspace` is similar to `arrange`. The difference is how we will request the number of samples that are required. With `linspace`, we can say how many elements we need between the given range. By default, it returns 50 elements. However, in `arrange`, we will need to specify the step size:

```
created_array = np.logspace(1,10,base=10.0)  
display_shape(created_array)
```

NumPy provides you with several functions to create special types of arrays:

```
ones_matrix = np.ones((3,3))  
display_shape(ones_matrix)  
  
# Create a matrix with all elements as 0  
zeros_matrix = np.zeros((3,3))  
display_shape(zeros_matrix)
```

The `ones()` and `zeros()` functions are used to create a matrix with 1 and 0 respectively:

```
[[ 1.  1.  1.]  
 [ 1.  1.  1.]  
 [ 1.  1.  1.]]  
  
Nuber of elements in a = 9  
Number of dimensions in a = 2  
Rows and Columns in a (3, 3)  
  
[[ 0.  0.  0.]  
 [ 0.  0.  0.]  
 [ 0.  0.  0.]]  
  
Nuber of elements in a = 9  
Number of dimensions in a = 2  
Rows and Columns in a (3, 3)
```

Identify that the matrices are created, as follows:

```
identity_matrix = np.eye(N=3,M=3,k=0)  
display_shape(identity_matrix)
```

The `k` parameter controls the index where value `1` has to start:

```
identity_matrix = np.eye(N=3,k=1)
display_shape(identity_matrix)
```

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]

Nuber of elements in a = 9
Number of dimensions in a = 2
Rows and Columns in a (3, 3)

[[ 0.  1.  0.]
 [ 0.  0.  1.]
 [ 0.  0.  0.]]

Nuber of elements in a = 9
Number of dimensions in a = 2
Rows and Columns in a (3, 3)
```

The shape of the arrays can be controlled by the `reshape` function:

```
# Array shaping
a_matrix = np.arange(9).reshape(3,3)
display_shape(a_matrix)
```

By passing `-1`, we can reshape the array to as many dimensions as needed:

```
# Paramter -1 refers to as many as dimension needed
back_to_array = a_matrix.reshape(-1)
display_shape(back_to_array)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]

Nuber of elements in a = 9
Number of dimensions in a = 2
Rows and Columns in a (3, 3)

[0 1 2 3 4 5 6 7 8]

Nuber of elements in a = 9
Number of dimensions in a = 1
Rows and Columns in a (9,)
```

The `ravel` and `flatten` functions can be used to convert a matrix to a one-dimensional array:

```
a_matrix = np.arange(9).reshape(3,3)
back_array = np.ravel(a_matrix)
display_shape(back_array)

a_matrix = np.arange(9).reshape(3,3)
back_array = a_matrix.flatten()
display_shape(back_array)
```

```
[0 1 2 3 4 5 6 7 8]

Nuber of elements in a = 9
Number of dimensions in a = 1
Rows and Columns in a  (9,)

[0 1 2 3 4 5 6 7 8]

Nuber of elements in a = 9
Number of dimensions in a = 1
Rows and Columns in a  (9,)
```

Let's look at some matrix operations, such as addition:

```
c_matrix = a_matrix + b_matrix
```

We will also look at element-wise multiplication:

```
d_matrix = a_matrix * b_matrix
```

The following code shows a matrix multiplication operation:

```
e_matrix = np.dot(a_matrix,b_matrix)
```

Finally, we will transpose a matrix:

```
f_matrix = e_matrix.T
```

The `min` and `max` functions can be used to find the minimum and maximum elements in a matrix. The `sum` function can be used to find the sum of the rows or columns in a matrix:

```
print
print "f_matrix,minimum = %d"%(f_matrix.min())
print "f_matrix,maximum = %d"%(f_matrix.max())
print "f_matrix, col sum",f_matrix.sum(axis=0)
print "f_matrix, row sum",f_matrix.sum(axis=1)
```

```
f_matrix,minimum = 15  
f_matrix,maximum = 111  
f_matrix, col sum [ 54 162 270]  
f_matrix, row sum [126 162 198]
```

The elements of a matrix can be reversed in the following way:

```
# reversing elements  
display shape(f matrix[::-1])
```

The `copy` function can be used to copy a matrix, as follows:

```
# Like python all elements are used by reference
# if copy is needed copy() command is used
f copy = f matrix.copy()
```

Finally, let's look at the `mgrid` functionality:

```
# Grid commands
xx,yy,zz = np.mgrid[0:3,0:3,0:3]
xx = xx.flatten()
yy = yy.flatten()
zz = zz.flatten()
```

The `mgrid` functionality can be used to get the coordinates in the m-dimension. In the preceding example, we have three dimensions. In each dimension, our values range from 0 to 3. Let us print `xx`, `yy`, and `zz` to understand a bit more:

```
[0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2]  
[0 0 0 1 1 1 2 2 2 0 0 0 1 1 1 2 2 2 2 0 0 0 1 1 1 2 2 2]  
[0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
```

Let's see the first element of each array. `[0,0,0]` is the first coordinate in our three-dimensional space. The second element in all three arrays, `[0,0,1]` is another point in our space. Similarly, using `mgrid`, we captured all the points in our three-dimensional coordinate system.

NumPy provides us with a module called `random` in order to give routines, which can be used to generate random numbers. Let's look at some examples of random number generation:

```
# Random numbers
general_random_numbers = np.random.randint(1,100, size=10)
print general_random_numbers
```

Using the `randint` function in the `random` module, we can generate random integer numbers. We can pass the `start`, `end`, and `size` parameters. In our case, our start is 1, our end is 100, and our size is 10. We want 10 random integers between 1 and 100. Let's look at the output that is returned:

```
[67 3 93 69 98 43 10 17 9 89]
```

Random numbers from other distributions can also be produced. Let's see an example where we get 10 random numbers from a `normal` distribution:

```
uniform_rnd_numbers = np.random.normal(loc=0.2,scale=0.2,size=10)
print uniform_rnd_numbers
```

Using the `normal` function, we will generate a random sample from a `normal` distribution. The mean and standard deviation parameters of the `normal` distribution are specified by the `loc` and `scale` parameters. Finally, `size` determines the number of samples.

By passing a tuple with the row and column values, we can generate a random matrix as well:

```
uniform_rnd_numbers = np.random.normal(loc=0.2,scale=0.2,size=(3, 3))
```

In the preceding example, we generated a 3 x 3 matrix, which is shown in the following code:

```
>>> print uniform_rnd_numbers
[ 0.29461598 -0.12032348 -0.19104886  0.16927785 -0.01208029  0.2303851
 0.2124355   0.20098306  0.05638245  0.06696319]
>>>
```

There's more...

You can refer to the following link for some excellent NumPy documentation:

<http://www.numpy.org/>

See also

- ▶ *Plotting with matplotlib* recipe in *Chapter 3, Analyzing Data - Explore & Wrangle*
- ▶ *Machine Learning with Scikit Learn* recipe in *Chapter 3, Analyzing Data - Explore & Wrangle*

Plotting with matplotlib

Matplotlib Python is a two-dimensional plotting library. All kinds of plots, including histograms, scatter plots, line plots, dot plots, heat maps, and others, can be generated by Python. In this book, we will use the `pyplot` interface of `matplotlib` for all our visualization requirements.

Getting ready

In this recipe, we will introduce basic plotting mechanisms using `pyplot`. We will use `pyplot` in almost all our recipes for visualization in this book.

We used matplotlib version 1.3.1 for all the recipes in this book. In your command line, you can invoke the `__version__` attribute to check for the version:

```
>>> matplotlib.__version__
'1.3.1'
```

How to do it...

Let's start by looking at how to plot simple graphs using matplotlib's `pyplot` module:

```
#Recipe_2a.py
import numpy as np
import matplotlib.pyplot as plt
def simple_line_plot(x,y,figure_no):
    plt.figure(figure_no)
    plt.plot(x,y)
    plt.xlabel('x values')
    plt.ylabel('y values')
    plt.title('Simple Line')

def simple_dots(x,y,figure_no):
    plt.figure(figure_no)
    plt.plot(x,y,'or')
    plt.xlabel('x values')
    plt.ylabel('y values')
    plt.title('Simple Dots')

def simple_scatter(x,y,figure_no):
    plt.figure(figure_no)
    plt.scatter(x,y)
    plt.xlabel('x values')
    plt.ylabel('y values')
```

```
plt.title('Simple scatter')

def scatter_with_color(x,y,labels,figure_no):
    plt.figure(figure_no)
    plt.scatter(x,y,c=labels)
    plt.xlabel('x values')
    plt.ylabel('y values')
    plt.title('Scatter with color')

if __name__ == "__main__":
    plt.close('all')
    # Sample x y data for line and simple dot plots
    x = np.arange(1,100,dtype=float)
    y = np.array([np.power(xx,2) for xx in x])

    figure_no=1
    simple_line_plot(x,y,figure_no)
    figure_no+=1
    simple_dots(x,y,figure_no)

    # Sample x,y data for scatter plot
    x = np.random.uniform(size=100)
    y = np.random.uniform(size=100)

    figure_no+=1
    simple_scatter(x,y,figure_no)
    figure_no+=1
    label = np.random.randint(2,size=100)
    scatter_with_color(x,y,label,figure_no)
    plt.show()
```

Now we will proceed to look at some advanced topics, including generating heat maps and labeling the x and y axes:

```
#Recipe_2b.py
import numpy as np
import matplotlib.pyplot as plt
def x_y_axis_labeling(x,y,x_labels,y_labels,figure_no):
    plt.figure(figure_no)
    plt.plot(x,y,'+r')
    plt.margins(0.2)
    plt.xticks(x,x_labels,rotation='vertical')
```

```
plt.yticks(y,y_labels,)

def plot_heat_map(x,figure_no):
    plt.figure(figure_no)
    plt.pcolor(x)
    plt.colorbar()

if __name__ == "__main__":
    plt.close('all')
    x = np.array(range(1,6))
    y = np.array(range(100,600,100))
    x_label = ['element 1','element 2','element 3','element
4','element 5']
    y_label = ['weight1','weight2','weight3','weight4','weight5']

    x_y_axis_labeling(x,y,x_label,y_label,1)

    x = np.random.normal(loc=0.5,scale=0.2,size=(10,10))
    plot_heat_map(x,2)

    plt.show()
```

How it works...

We will start by importing the required modules. While using `pyplot`, it's recommended that you import NumPy:

```
import numpy as np
import matplotlib.pyplot as plt
```

Let's start by following the code from the main function. There may be graphs from the previously run program. It is good practice to close them all, as we will use more graphs in our program:

```
plt.close('all')
```

We will proceed by generating some data using NumPy to demonstrate plotting using `pyplot`:

```
# Sample x y data for line and simple dot plots
x = np.arange(1,100,dtype=float)
y = np.array([np.power(xx,2) for xx in x])
```

We generated 100 elements in both our x and y variables. Our y is a square of our x variable.

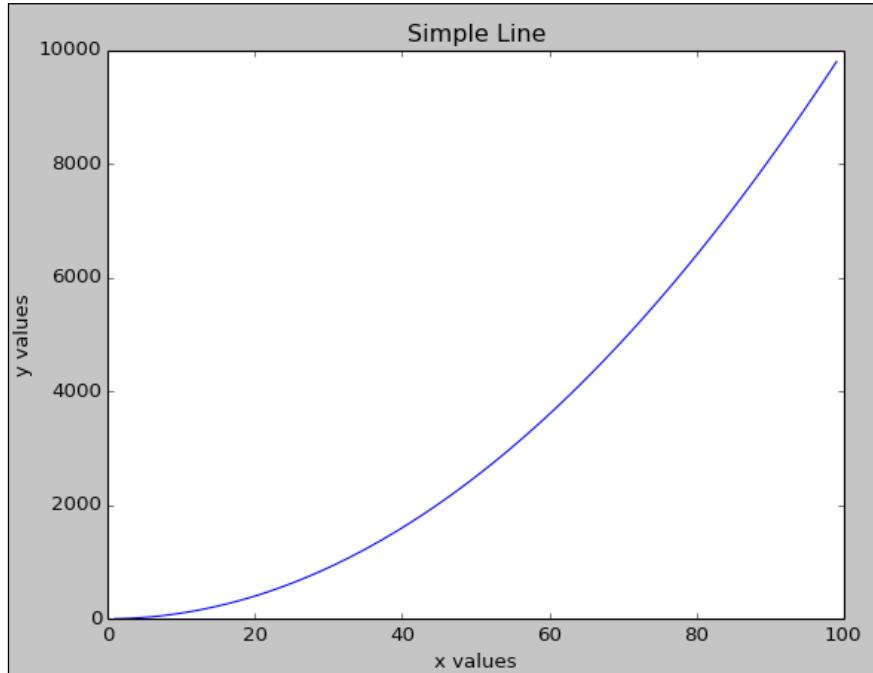
Let's proceed to doing a simple line plot:

```
figure_no=1  
simple_line_plot(x,y,figure_no)
```

When our program has multiple plots, it's a good practice to number each plot. Variable `figure_no` is used to number our plots. Let's look at the `simple_line_plot` function:

```
def simple_line_plot(x,y,figure_no):  
    plt.figure(figure_no)  
    plt.plot(x,y)  
    plt.xlabel('x values')  
    plt.ylabel('y values')  
    plt.title('Simple Line')
```

As you can see, we started numbering our plots by calling the `figure` function in `pyplot`. We passed the `figure no` variable from our main program. After this, we simply called the `plot` function with our x and y values. We can make our plot meaningful by giving names to our x and y axes using the `xlabel` and `ylabel` functions respectively. Finally, we can also give a title to our plot. That is it. Our first simple line plot is ready. The plot will not be displayed till the `show()` function is called. In our code, we will invoke the `show()` function in order to see all the plots together. Our plot will look as follows:



Here, we plotted the values x on the x axis and x squared on the y axis.

We created a simple line plot. We can see a nice curve as our **y values** are squares of our **x values**.

Let's move on to our next plot:

```
figure_no+=1  
simple_dots(x,y,figure_no)
```

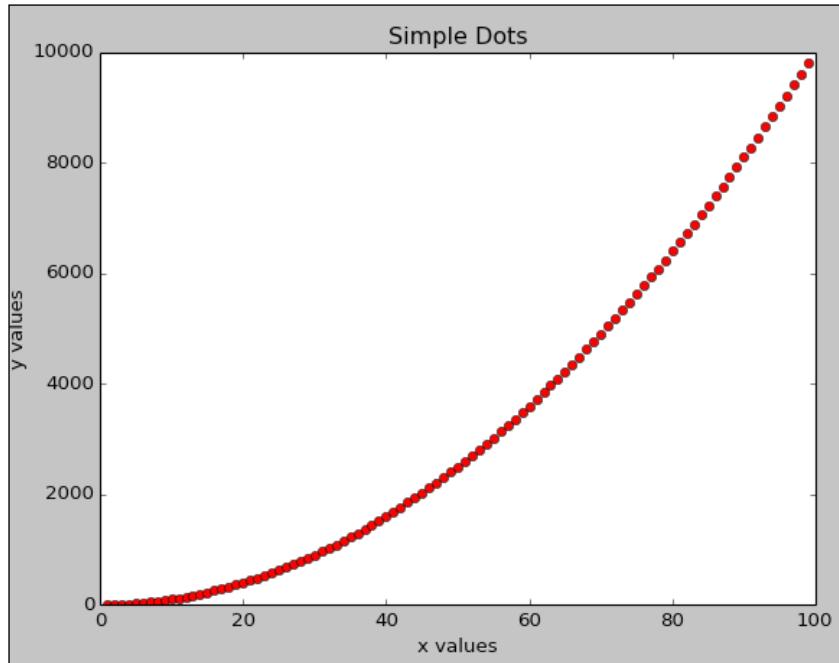
We will increment our figure number and call the `simple_dots` function. We want to plot our x and y values as dots instead of a line. Let's look at the `simple_dots` function:

```
def simple_dots(x,y,figure_no):  
    plt.figure(figure_no)  
    plt.plot(x,y,'or')  
    plt.xlabel('x values')  
    plt.ylabel('y values')  
    plt.title('Simple Dots')
```

Every line is similar to our previous function except the following line:

```
plt.plot(x,y,'or')
```

The `or` parameter says that we need dots (\circ), and the dots should be in the color red (r). The following is the output of the preceding command:



Let's move to our next plot.

We are going to see a scatter plot. Let's generate some data using NumPy:

```
# Sample x,y data for scatter plot
x = np.random.uniform(size=100)
y = np.random.uniform(size=100)
```

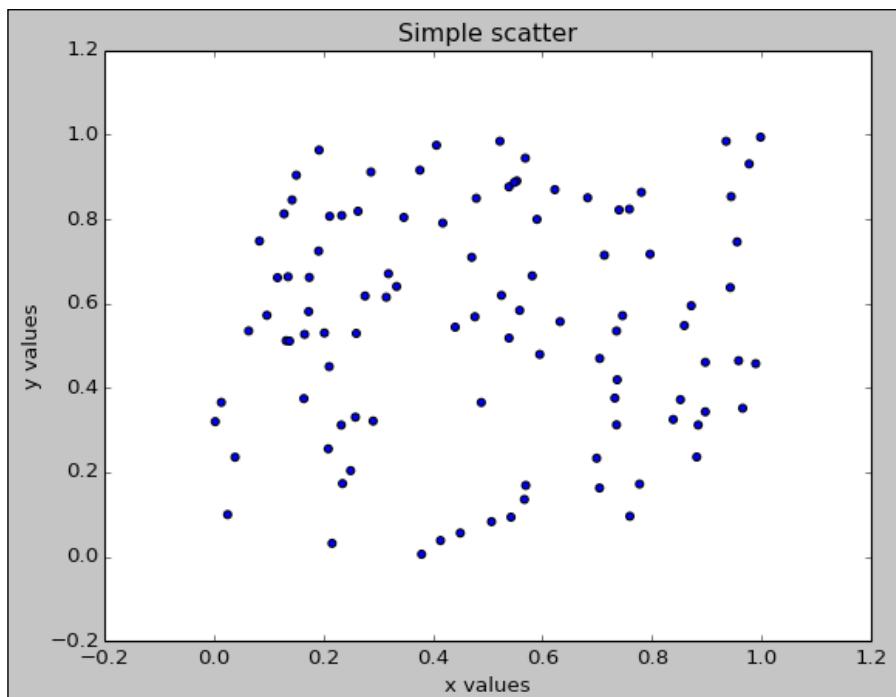
We sampled 100 data points from a uniform distribution. Now we will proceed to call the `simple_scatter` function in order to generate our scatter plot:

```
figure_no+=1
simple_scatter(x,y,figure_no)
```

In the `simple_scatter` function, all the lines are similar to the previous plotting routines except for the following line:

```
plt.scatter(x,y)
```

Instead of calling the `plot` function in `pyplot`, we invoked the `scatter` function. Our plot will look as follows:



Let's move on to our final plot, which is a scatter plot, but the points are colored based on the class label that they belong to:

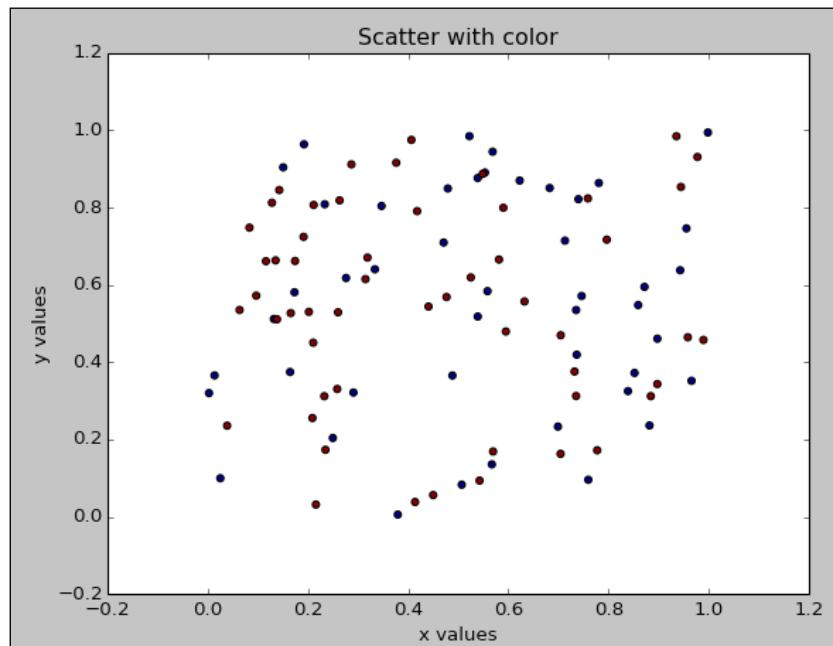
```
figure_no+=1  
label = np.random.randint(2, size=100)  
scatter_with_color(x,y,label,figure_no)
```

We will increment our figure in order to keep track of our graph. In the next line, we will assign some random labels, either 1 or 0, to our points. Finally, we will call the `scatter_with_color` function with our `x`, `y`, and `label` variables.

In the function, let's look at the line that differentiates this code from the previous scatter plot code:

```
plt.scatter(x,y,c=labels)
```

We will pass our labels to a `c` parameter, which stands for color. Each label will be assigned a unique color. In our example, all the points that are labeled as 0 will get a color that is different from the points that are labeled as 1, as follows:



Let's move on to plotting some heat maps, and axis labeling.

Once again, we will start with the main function:

```
plt.close('all')
x = np.array(range(1,6))
y = np.array(range(100,600,100))
x_label = ['element 1','element 2','element 3','element
4','element 5']
y_label = ['weight1','weight2','weight3','weight4','weight5']

x_y_axis_labeling(x,y,x_label,y_label,1)
```

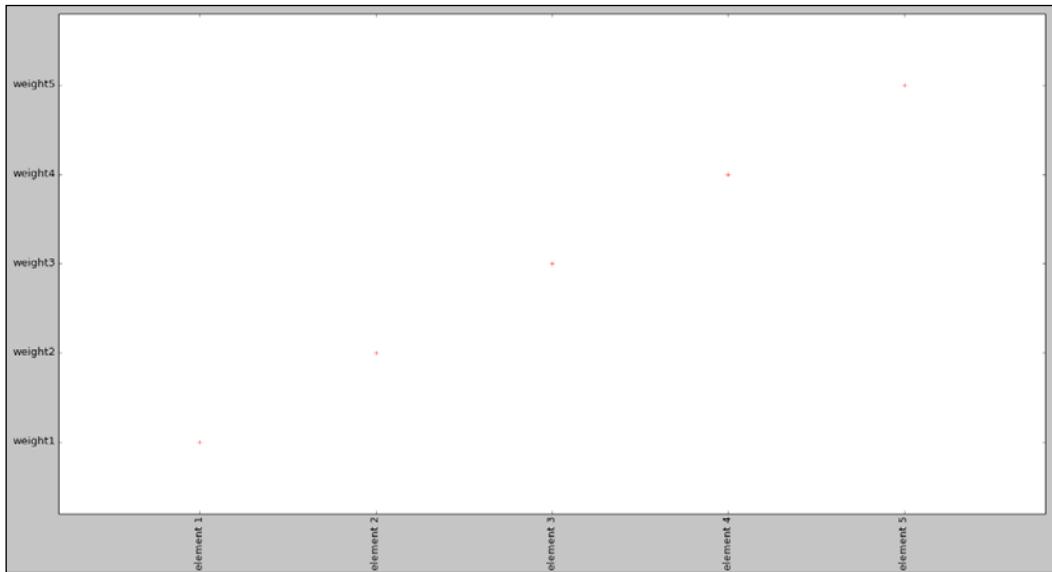
As a good practice, we will close all the previous figures by calling the `close` function. We will proceed with generating some data. Our `x` is an array of five elements, starting from 1 and ending with 5. Our `y` is an array of five elements, starting from 100 and ending with 500. We defined the two `x_label` and `y_label` lists, which will serve as the labels for our plot. Finally, we invoked the `x_y_axis_labeling` function in order to demonstrate the concept of labeling our tickers in the `x` and `y` axes.

Let's look at the following function:

```
def x_y_axis_labeling(x,y,x_labels,y_labels,figure_no):
    plt.figure(figure_no)
    plt.plot(x,y,'+r')
    plt.margins(0.2)
    plt.xticks(x,x_labels,rotation='vertical')
    plt.yticks(y,y_labels,)
```

We will do a simple dot plot by calling pyplot's `dot` function. However, in this case, we want our points to be displayed as `+` instead of `o`. Hence, we will specify `+r`. Our color of choice is red, hence `r`.

In the next two lines, we will specify what our `x` axis and `y` axis tickers need to be. By calling the `xticks` function, we will pass on our `x` values and their labels. In addition, we will say that we want the text to be rotated vertically so that they don't overlap each other. Similarly, we will specify the tickers for the `y` axis. Let's look at our plot, as follows:



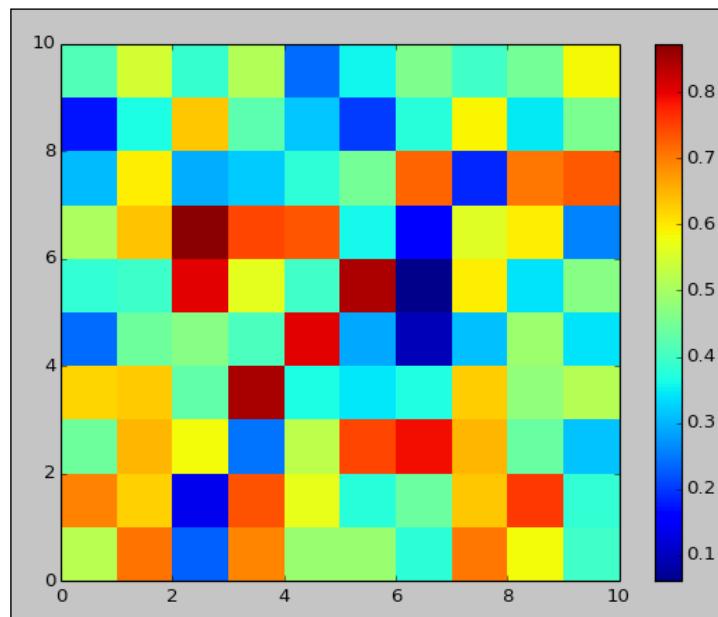
Let's see how to generate heat maps using pyplot:

```
x = np.random.normal(loc=0.5,scale=0.2,size=(10,10))
plot_heat_map(x,2)
```

We will generate some data for our heat map. In this case, we generated a 10×10 matrix filled with values from a normal distribution of a mean specified by a `loc` variable of 0.5 and standard deviation specified by a `scale` variable of 0.2. We will invoke the `plot_heat_map` function with this matrix. The second parameter is the figure number:

```
def plot_heat_map(x,figure_no):
    plt.figure(figure_no)
    plt.pcolor(x)
    plt.colorbar()
```

We will call the `pcolor` function in order to generate a heat map. The next line invokes the `colorbar` function to display the color gradients for our range of values:



There's more...

For more information on matplotlib, you can refer to the general matplotlib documentation at http://matplotlib.org/faq/usage_faq.html.

The following link is an excellent tutorial on pyplot:

http://matplotlib.org/users/pyplot_tutorial.html

Matplotlib provides excellent three-dimensional plotting capabilities. Refer to the following link for more information:

http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html

The pylab module in matplotlib combines the name space of NumPy with pyplot. Pylab can also be used to generate the various types of plots shown in this recipe.

Machine learning with scikit-learn

Scikit-learn is a versatile machine learning library in Python. We will use this library extensively in this book. We used scikit-learn version 0.15.2 for all the recipes in this book. In the command line, you can invoke the `__version__` attribute to check for the version:

```
>>> sklearn.__version__
'0.15.2'
>>> |
```

Getting ready

In this recipe, we will demonstrate some of the capabilities of scikit-learn and learn about some of their API organization so that we can follow it seamlessly in our future recipes.

How to do it...

Scikit-learn provides us with an inbuilt dataset. Let's see how to access this dataset and use it:

```
#Recipe_3a.py
from sklearn.datasets import load_iris,load_boston,make_classification
make_circles, make_moons

# Iris dataset
data = load_iris()
x = data['data']
y = data['target']
y_labels = data['target_names']
x_labels = data['feature_names']

print
print x.shape
print y.shape
print x_labels
print y_labels

# Boston dataset
data = load_boston()
x = data['data']
y = data['target']
x_labels = data['feature_names']
```

```
print
print x.shape
print y.shape
print x_labels

# make some classification dataset
x,y = make_classification(n_samples=50,n_features=5, n_classes=2)

print
print x.shape
print y.shape

print x[1,:]
print y[1]

# Some non linear dataset
x,y = make_circles()
import numpy as np
import matplotlib.pyplot as plt
plt.close('all')
plt.figure(1)
plt.scatter(x[:,0],x[:,1],c=y)

x,y = make_moons()
import numpy as np
import matplotlib.pyplot as plt
plt.figure(2)
plt.scatter(x[:,0],x[:,1],c=y)

plt.show()
```

Let's proceed with seeing how we can invoke some machine learning functionalities in scikit-learn:

```
#Recipe_3b.py
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
# Data Preprocessing routines
x = np.asmatrix([[1,2],[2,4]])
poly = PolynomialFeatures(degree = 2)
poly.fit(x)
x_poly = poly.transform(x)

print "Original x variable shape",x.shape
```

```
print x
print
print "Transformed x variables",x_poly.shape
print x_poly

#alternatively
x_poly = poly.fit_transform(x)

from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris

data = load_iris()
x = data['data']
y = data['target']

estimator = DecisionTreeClassifier()
estimator.fit(x,y)
predicted_y = estimator.predict(x)
predicted_y_prob = estimator.predict_proba(x)
predicted_y_lprob = estimator.predict_log_proba(x)

from sklearn.pipeline import Pipeline

poly = PolynomialFeatures(n=3)
tree_estimator = DecisionTreeClassifier()

steps = [('poly',poly),('tree',tree_estimator)]
estimator = Pipeline(steps=steps)
estimator.fit(x,y)
predicted_y = estimator.predict(x)
```

How it works...

Let's load the scikit learn library and import the module that contains the various functions in order to extract the inbuilt datasets:

```
from sklearn.datasets import load_iris,load_boston,make_classification
```

The first dataset that we will look at is the iris dataset. Refer to https://en.wikipedia.org/wiki/Iris_flower_data_set for more information.

Introduced by Sir Donald Fisher, this is a classic dataset for a classification problem:

```
data = load_iris()
x = data['data']
y = data['target']
y_labels = data['target_names']
x_labels = data['feature_names']
```

The `load_iris` function, when invoked, returns a dictionary object. The predictor `x`, response variable `y`, response variable names, and feature names can be extracted by querying the dictionary object with the appropriate keys.

Let's proceed to print them and see their values:

```
print
print x.shape
print y.shape
print x_labels
print y_labels
```

```
(150, 4)
(150,)
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
['setosa' 'versicolor' 'virginica']
```

As you can see, our predictors have 150 instances and four attributes. Our response variable has 150 instances and a class label for each of the rows in our predictor set. We will then print out the attribute names, petal and sepal width and length, and finally, the class labels. In most of our future recipes, we will use this dataset extensively.

Let's proceed to inspect another inbuilt dataset called the Boston housing dataset used in a regression problem:

```
# Boston dataset
data = load_boston()
x = data['data']
y = data['target']
x_labels = data['feature_names']
```

The data is loaded pretty much the same as was `iris`, and the various components of the data, including the predictors and response variables, are queried using the respective keys from the dictionary. Let's print these variables in order to inspect them:

```
(506, 13)
(506,)
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

As you can see, our predictor set `x` has 506 instances and 13 attributes. Our response variable has 506 entries. Finally, we will also print out the names of our attributes.

Scikit-learn also provides us with functions that will help us produce a random classification dataset with some desired properties:

```
# make some classification dataset
x,y = make_classification(n_samples=50,n_features=5, n_classes=2)
```

The `make_classification` function is a function that can be used to generate a classification dataset. In our example, we generated a dataset with 50 instances that are dictated by the `n_samples` parameter, five attributes, `n_features` parameters, and two classes set by the `n_classes` parameter. Let's inspect the output of this function:

```
print x.shape
print y.shape

print x[1,:]
print y[1]
```

```
(50, 5)
(50,)
[ 1.09036697 -0.00209392 -1.85449661 -0.81583736 -0.3623406 ]
1
```

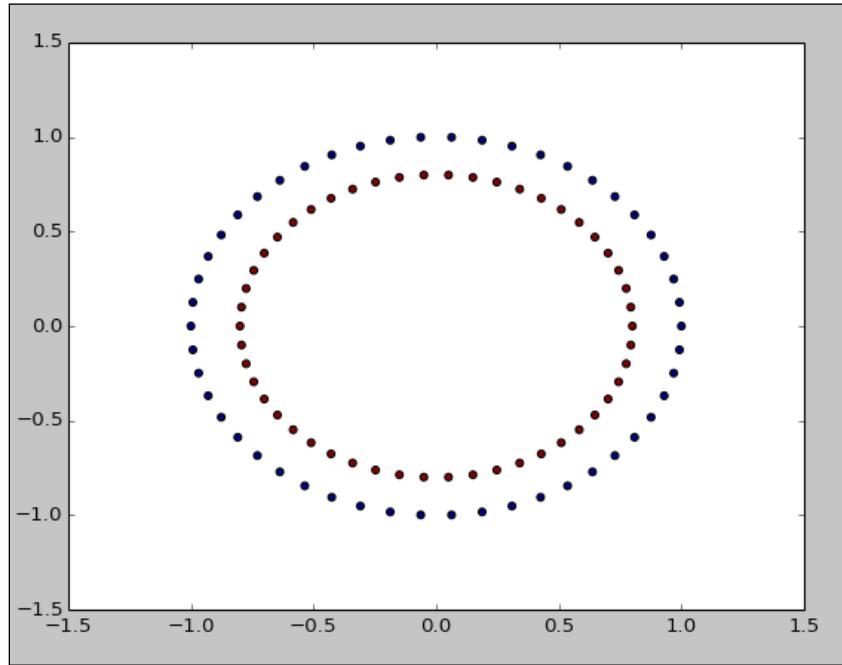
As you can see, our predictor `x` has 150 instances with five features. Our response variable has 150 instances, with a class label for each of the prediction instances.

We will print out the second record in our predictor set, `x`. You can see that we have a vector of dimension 5, relating to the five features that we requested. Finally, we will also print the response variable, `y`. For the second row of our predictors, the class label is 1.

Scikit-learn also provides us with the functions that can generate data with nonlinear relationships:

```
# Some non linear dataset
x,y = make_circles()
import numpy as npimport matplotlib.pyplot as plt
plt.close('all')
plt.figure(1)
plt.scatter(x[:,0],x[:,1],c=y)
```

You should be familiar with `pyplot` now from the previous recipe. Let's see our plot first to understand the nonlinear relationship:

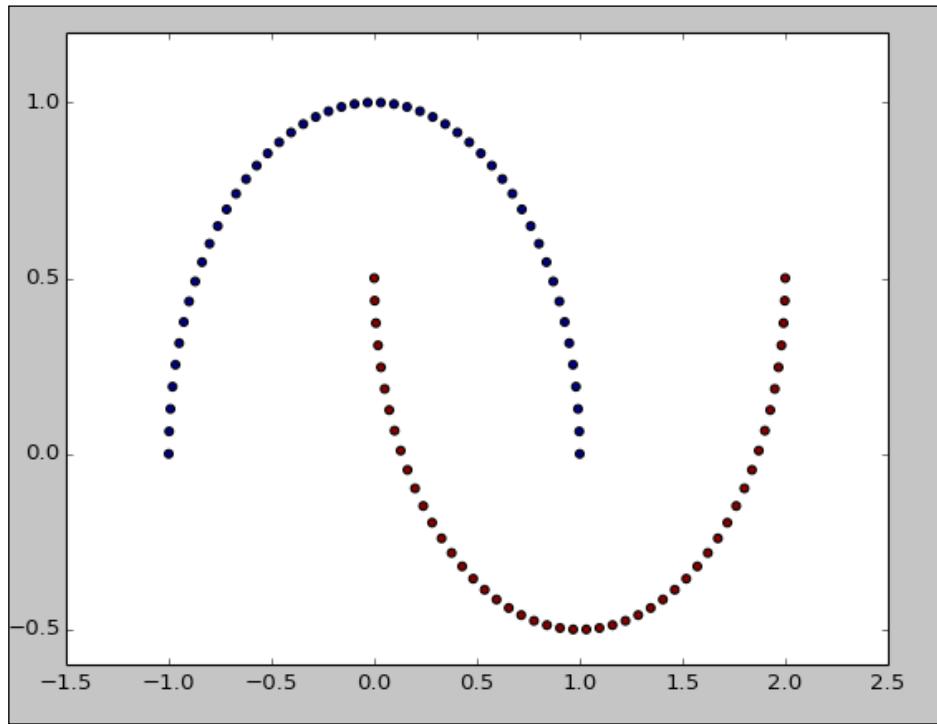


As you can see, our classification has produced two concentric circles. Our `x` is a dataset with two variables. Variable `y` is the class label. As shown by the concentric circle, the relationship between our prediction variable is nonlinear.

Another interesting function to produce a nonlinear relationship is `make_moons` from `scikit-learn`:

```
x,y = make_moons()  
import numpy as np  
import matplotlib.pyplot as plt  
plt.figure(2)  
plt.scatter(x[:,0],x[:,1],c=y)
```

Let's look at its plot in order to understand the nonlinear relationship:



The crescent-shaped plot shows that the attributes in our predictor set x are nonlinearly related to each other.

Let's switch gears to understand the API structure of scikit-learn. One of the major advantages of using scikit-learn is its clean API structure. All the data modeling classes deriving from the `BaseEstimator` class have to strictly implement the `fit` and `transform` functions. We will see some examples to learn more about this.

Let's start with the preprocessing module in scikit-learn:

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
```

We will use the `PolynomialFeatures` class in order to demonstrate the ease of using scikit-learn's SDK. Refer to the following link for polynomials:

<https://en.wikipedia.org/wiki/Polynomial>

With a set of predictor variables, we may want to add some more variables to our predictor set in order to see if our model accuracy can be improved. We can use the polynomials of the existing features as a new feature. The `PolynomialFeatures` class helps us do this:

```
# Data Preprocessing routines
x = np.asmatrix([[1,2],[2,4]])
```

We will first create a dataset. In this case, our dataset has two instances and two attributes:

```
poly = PolynomialFeatures(degree = 2)
```

We will proceed to instantiate our `PolynomialFeatures` class with the required degree of polynomials. In this case, it will be a second degree:

```
poly.fit(x)
x_poly = poly.transform(x)
```

Then, there are two functions, `fit` and `transform`. The `fit` function is used to do the necessary calculations for the transformation. In this case, `fit` is redundant, but we will see some more examples of how `fit` is used later in this recipe.

The `transform` function takes the input and, based on the calculations performed by `fit`, transforms the given input:

```
#alternatively
x_poly = poly.fit_transform(x)
```

Alternatively, in this case, `fit` and `transform` can be called in one shot. Let's look at the value and shape of our original and transformed `x` variable:

<code>Original x variables</code>
<code>[[1 2]</code>
<code>[2 4]]</code>
<code>Transformed x variables</code>
<code>[[1 1 2 1 2 4]</code>
<code>[1 2 4 4 8 16]]</code>

Any class that implements a machine learning method in scikit-learn has to deliver from `BaseEstimator`. See the following link for `BaseEstimator`:

<http://scikit-learn.org/stable/modules/generated/sklearn.base.BaseEstimator.html>

`BaseEstimator` expects that the implementation class provides both the `fit` and `transform` methods. This way the API is kept very clean.

Let's see another example. Here, we imported a class called `DecisionTreeClassifier` from the module `tree`. `DecisionTreeClassifier` implements the decision tree algorithm:

```
from sklearn.tree import DecisionTreeClassifier  
Let's put this class into action:
```

```
from sklearn.datasets import load_iris  
  
data = load_iris()  
x = data['data']  
y = data['target']  
  
estimator = DecisionTreeClassifier()  
estimator.fit(x,y)  
predicted_y = estimator.predict(x)  
predicted_y_prob = estimator.predict_proba(x)  
predicted_y_lprob = estimator.predict_log_proba(x)
```

Let's use the iris dataset to see how the tree algorithm can be used. We will load the iris dataset in the `x` and `y` variables. We will then instantiate `DecisionTreeClassifier`. We will proceed to build the model by invoking the `fit` function and passing our `x` predictor and `y` response variable. This will build the tree model. Now, we are ready with our model to do some predictions. We will use the `predict` function in order to predict the class labels for the given input. As you can see, we leveraged the same `fit` and `predict` method as in `PolynomialFeatures`. There are two other methods, `predict_proba`, which gives the probability of the prediction, and `predict_log_proba`, which provides the logarithm of the prediction probability.

Let's now see another interesting utility called pipe lining. Various machine learning methods can be chained together using pipe lining:

```
from sklearn.pipeline import Pipeline  
  
poly = PolynomialFeatures(n=3)  
tree_estimator = DecisionTreeClassifier()
```

Let's start by instantiating the data processing routines, `PolynomialFeatures` and `DecisionTreeClassifier`:

```
steps = [('poly',poly),('tree',tree_estimator)]
```

We will define a list of tuples to indicate the order of our chaining. We want to run the polynomial feature generation, followed by our decision tree:

```
estimator = Pipeline(steps=steps)  
estimator.fit(x,y)  
predicted_y = estimator.predict(x)
```

We can now instantiate our Pipeline object with the list declared using the steps variable. Now, we can proceed to do business as usual by calling the `fit` and `predict` methods.

We can invoke the `named_steps` attribute in order to inspect the models in the various stages of our pipeline:

```
>>> estimator.named_steps
{'tree': DecisionTreeClassifier(compute_importances=None, criterion='gini',
                                 max_depth=None, max_features=None, max_leaf_nodes=None,
                                 min_density=None, min_samples_leaf=1, min_samples_split=2,
                                 random_state=None, splitter='best'), 'poly': PolynomialFeatures(degree=3, include_bias=True, interaction_only=False)}
```

There's more...

There are a lot more dataset creation functions available in scikit-learn. Refer to the following link:

<http://scikit-learn.org/stable/datasets/>

While creating nonlinear datasets using `make_circle` and `make_moons`, we mentioned that a lot of desired properties can be added to the dataset. The data can be corrupted slightly by inducing incorrect class labels. Refer to the following link for a list of options that are available in order to introduce such nuances in the data:

http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html

http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html

See also

- ▶ *Plotting recipe in Chapter 2, Working with Python Environments*

3

Data Analysis – Explore and Wrangle

We will cover the following recipes in this chapter:

- ▶ Analyzing univariate data graphically
- ▶ Grouping the data and using dot plots
- ▶ Using scatter plots for multivariate data
- ▶ Using heat maps
- ▶ Performing summary statistics and plots
- ▶ Using a box-and-whisker plot
- ▶ Imputing the data
- ▶ Performing random sampling
- ▶ Scaling the data
- ▶ Standardizing the data
- ▶ Performing tokenization
- ▶ Removing stop words
- ▶ Stemming the words
- ▶ Performing word lemmatization
- ▶ Representing the text as a bag of words
- ▶ Calculating term frequencies and inverse document frequencies

Introduction

Before you venture into any data science application, it is always helpful in the long run to have a good understanding of the data that you are about to process. An understanding of the underlying data will help you choose the right algorithm to use for the problem at hand. Exploring the data at various levels of granularity is called **Exploratory Data Analysis (EDA)**. In many cases, **EDA** can uncover patterns that are typically revealed by a data mining algorithm. **EDA** helps us understand data characteristics and provides you with the proper guidance in order to choose the right algorithm for the given problem.

In this chapter, we will cover **EDA** in detail. We will look into the practical techniques and tools that are used to perform **EDA** operations in an effective way.

Data preprocessing and transformation are two other important processes that can improve the quality of data science models and increase the success rate of data science projects.

Data preprocessing is the process of making the data ready in order to be ingested either by a data mining method or machine learning algorithm. It encompasses many things such as data cleaning, attribute subset selection, data transformation, and others. We will cover both numerical data preprocessing and text data preprocessing in this chapter.

Text data is a different beast than the numerical data. We need different transformation methods in order to make it suitable for ingestion in the machine learning algorithms. In this chapter, we will see how we can transform the text data. Typically, text transformation is a staged process with various components in the form of a pipeline.

Some of the components are as follows:

- ▶ Tokenization
- ▶ Stop word removal
- ▶ Base form conversion
- ▶ Feature derivation

Typically, these components are applied to a given text in order to extract features. At the end of the pipeline, the text data is transformed in a way that it can be fed as input to the machine learning algorithms. In this chapter, we will see recipes for every component listed in the preceding pipeline.

Many times, a lot of errors may be introduced during the data collection phase. These may be due to human errors, limitations, or bugs in the data measuring or collective process/device. Data inconsistency is a big challenge. We will start our data preprocessing journey with data imputation is a way to handle errors in the incoming data and then proceed to other methods.

Analyzing univariate data graphically

Datasets with only one variable/column are called univariate data. Univariate is a general term in mathematics, which refers to any expression, equation, function, or polynomial with only one variable. In our case, we will restrict the univariate function to datasets. Let's say that we will measure the heights of a group of people in meters; the data will look as follows:

5, 5.2, 6, 4.7,...

Our measurement is only about a single attribute of people, height. This is an example of univariate data.

Getting ready

Let's start our **EDA** recipe by looking at a sample univariate dataset through visualization. It is easy to analyze the data characteristics through the right visualization techniques. We will use `pyplot` to draw graphs in order to visualize the data. Pyplot is the state-machine interface to the `matplotlib` plotting library. Figures and axes are implicitly and automatically created to achieve the desired plot. The following link is a good reference for `pyplot`:

http://matplotlib.org/users/pyplot_tutorial.html

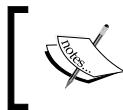
For this example, we will use a number of Presidential Requests of Congress in State of the Union Address. The following URL contains the data:

<http://www.presidency.ucsb.edu/data/sourequests.php>

The following is a sample of the data:

```
1946, 41
1947, 23
1948, 16
1949, 28
1950, 20
1951, 11
1952, 19
1953, 14
1954, 39
1955, 32
1956,
1957, 14
1958,
1959, 16
1960, 6
```

We will visually look at this data and identify any outliers present in the data. We will follow a recursive approach with respect to the outliers. Once we have identified the outliers, we will remove them from the dataset and plot the remaining data in order to find any new outliers.



Recursively looking into the data after removing the perceived outlier in every iteration is a common approach in detection of outliers.

How to do it...

We will load the data using NumPy's data loading utility. Then, we will address the data quality issues; in this case, we will address how to handle the null values. As you can see in the data, the years 1956 and 1958 have null entries. Let's replace the null values by 0 using the lambda function.

Following this, let's plot the data to look for any trends:

```
# Load libraries
import numpy as np
from matplotlib.pylab import frange
import matplotlib.pyplot as plt

fill_data = lambda x : int(x.strip() or 0)
data = np.genfromtxt('president.txt', dtype=(int,int), converters={1:fill_data}, \
                     delimiter=",")
x = data[:,0]
y = data[:,1]

# 2. Plot the data to look for any trends or values
plt.close('all')
plt.figure(1)
plt.title("All data")
plt.plot(x,y,'ro')
plt.xlabel('year')plt.ylabel('No Presidential Request')
```

Let's calculate the percentile values and plot them as references in the plot that has been generated:

```
#3.Calculate percentile values (25th, 50th,75th) for the data to
understand data distribution
perc_25 = np.percentile(y,25)
perc_50 = np.percentile(y,50)
perc_75 = np.percentile(y,75)
```

```
print
print "25th Percentile      = %0.2f"%(perc_25)
print "50th Percentile     = %0.2f"%(perc_50)
print "75th Percentile     = %0.2f"%(perc_75)
print
#4. Plot these percentile values as reference in the plot we generated
# in the previous step.
# Draw horizontal lines at 25,50 and 75th percentile
plt.axhline(perc_25,label='25th perc',c='r')
plt.axhline(perc_50,label='50th perc',c='g')
plt.axhline(perc_75,label='75th perc',c='m')plt.legend(loc='best')
```

Finally, let's inspect the data visually for outliers and then remove them using the mask function. Let's plot the data again without the outliers:

```
#5. Look for outliers if any in the data by visual inspection.
# Remove outliers using mask function
# Remove outliers 0 and 54
y_masked = np.ma.masked_where(y==0,y)
# Remove point 54
y_masked = np.ma.masked_where(y_masked==54,y_masked)

#6 Plot the data again.
plt.figure(2)
plt.title("Masked data")
plt.plot(x,y_masked,'ro')
plt.xlabel('year')
plt.ylabel('No Presedential Request')
plt.ylim(0,60)

# Draw horizontal lines at 25,50 and 75th percentile
plt.axhline(perc_25,label='25th perc',c='r')
plt.axhline(perc_50,label='50th perc',c='g')
plt.axhline(perc_75,label='75th perc',c='m')
plt.legend(loc='best')plt.show()
```

How it works...

In the first step, we will put some data loading techniques that we learnt in the previous chapter to action. You will have noticed that the years 1956 and 1958 are left blank. We will replace them with 0 using an anonymous function:

```
fill_data = lambda x : int(x.strip() or 0)
```

The `fill_data` lambda function will replace any null value in the dataset; in this case, line no 11 and 13 with 0:

```
data = np.genfromtxt('president.txt', dtype=(int,int), converters={1:fill_data}, delimiter=",")
```

We will pass `fill_data` to the `genfromtxt` function's `converters` parameter. Note that `converters` takes a dictionary as its input. The key in the dictionary dictates which column our function should be applied to. The value indicates the function. In this case, we specified `fill_data` as the function and set the key to 1 indicating that the `fill_data` function has to be applied to column 1. Now let's look at the data in the console:

```
>>> data[7:15]
array([[1953,    14],
       [1954,    39],
       [1955,    32],
       [1956,     0],
       [1957,    14],
       [1958,     0],
       [1959,    16],
       [1960,     6]])
```

```
>>>
```

As we can see, the years 1956 and 1958 have a 0 value added to them. For the ease of plotting, we will load the year data in `x` and the number of Presidential Requests to Congress in the State of Union Address to `y`:

```
x = data[:,0]
y = data[:,1]
```

As you can see, in the first column, the year is loaded in `x` and the next column in `y`.

In step 2, we will plot the data with the `x` axis as the year and `y` axis representing the values:

```
plt.close('all')
```

We will first close any previous graphs that are open from the previous programs:

```
plt.figure(1)
```

We will give a number to our plot. This is very useful when we have a lot of graphs in a program:

```
plt.title("All data")
```

We will specify a title for our plot:

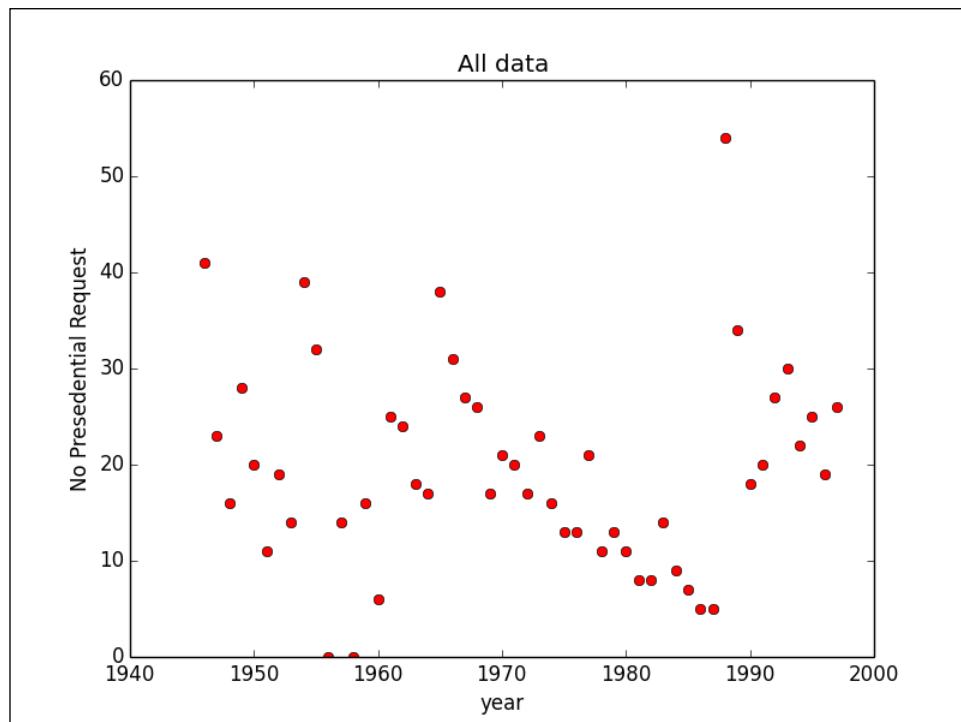
```
plt.plot(x,y, 'ro')
```

Finally, we will plot x and y. The 'ro' parameter tells pyplot to plot x and y as dots (0) in the color red (r):

```
plt.xlabel('year')
plt.ylabel('No Presidential Request')
```

Finally, the x and y axes labels are provided.

The output looks as follows:



A casual look at this graph shows that the data is spread everywhere and no trends or patterns can be found in the first glance. However, with a keen eye, you can notice three points: one point at the top on the right-hand side and others to the immediate left of **1960** in the x axis. They are starkly different from all the other points in the sample, and hence, they are outliers.



An outlier is an observation that lies outside the overall pattern of a distribution (Moore and McCabe 1999).

In order to understand these points further, we will take the help of percentiles.

If we have a vector V of length N, the qth percentile of V is the qth ranked value in a sorted copy of V. The values and distances of the two nearest neighbors as well as the *interpolation* parameter will determine the percentile if the normalized ranking does not match q exactly. This function is the same as the median if $q=50$, the same as the minimum if $q=0$, and the same as the maximum if $q=100$.

Refer to <http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.percentile.html> for more information.

Why don't we use averages? We will look into averages in the summary statistics section; however, looking at the percentiles has its own advantages. Average values are typically skewed by outliers; outliers such as the one at the top on the right-hand side can drag the average to a higher value and the outliers near 1960 can do the opposite. Percentiles give us a better clarity about the range of values in our dataset. We can calculate the percentiles using NumPy.

In step 3, we will calculate the percentiles and print them.

The percentile values calculated and printed for this dataset are as follows:

25th Percentile	= 13.00
50th Percentile	= 18.50
75th Percentile	= 25.25

Interpreting the percentiles:

25% of the points in the dataset are below 13.00 (25th percentile value).

50% of the points in the dataset are below 18.50 (50th percentile value).

75% of the points in the dataset are below 25.25 (75th percentile value).

A point to note is that the 50th percentile is the median. Percentiles give us a good idea of the range of our values.

In step 4, we will plot these percentile values as horizontal lines in our graph in order to enhance our visualization:

```
# Draw horizontal lines at 25,50 and 75th percentile
plt.axhline(perc_25,label='25th perc',c='r')
plt.axhline(perc_50,label='50th perc',c='g')
plt.axhline(perc_75,label='75th perc',c='m')
plt.legend(loc='best')
```

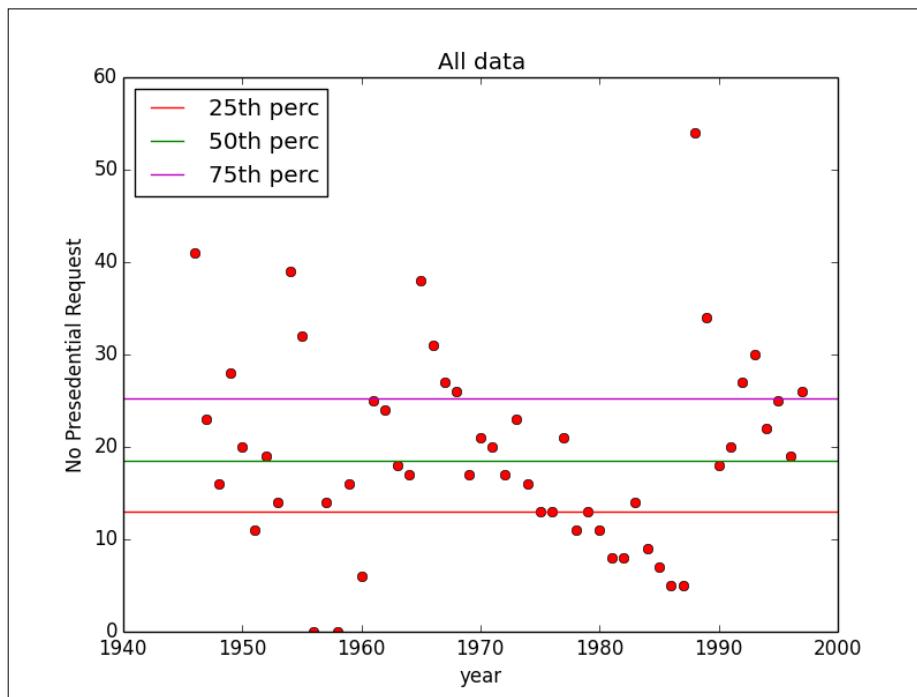
We used the `plt.axhline()` function to draw these horizontal lines. This function will draw a line at the given y value from the minimum of x to the maximum of x. Using the `label` parameter, we gave it a name and set the color of the line through the `c` parameter.



A good way to understand any function is to pass the function name to `help()` in the Python console. In this case, `help(plt.axhline)` in the Python console will give you the details.

Finally, we will place the legend using `plt.legend()`, and using the `loc` parameter, ask pyplot to determine the best location to put the legend so that it does not affect the plot readability.

Our graph is now as follows:

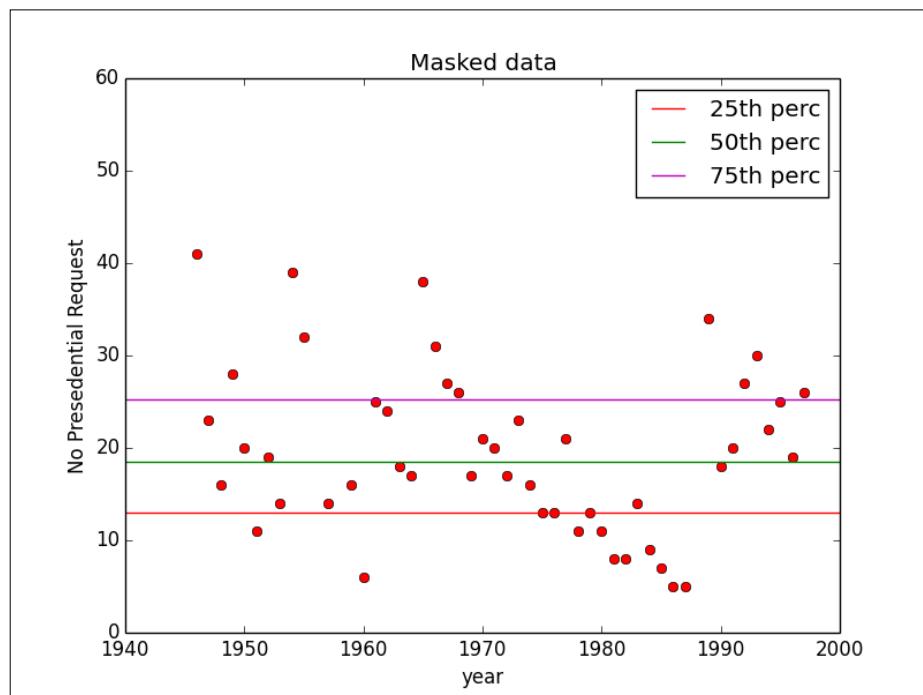


In step 5, we will move on to remove the outliers using the `mask` function in NumPy:

```
# Remove zero values
y_masked = np.ma.masked_where(y==0,y)
# Remove 54
y_masked = np.ma.masked_where(y_masked==54,y_masked)
```

Masking is a convenient way to hide some of the values without removing them from our array. We used the `ma.masked_where` function, where we passed a condition and an array. The function then masks the values in the array that meet the condition. Our first condition was to mask all the points in the `y` array, where the array value was 0. We stored the new masked array as `y_masked`. Then, we applied another condition on `y_masked` to remove point 54.

Finally, in step 6, we will repeat the plotting steps. Our final plot looks as follows:



See also

- ▶ [Creating Anonymous functions](#) recipe in *Chapter 1, Using Python for Data Science*
- ▶ [Pre-processing columns](#) recipe in *Chapter 1, Using Python for Data Science*
- ▶ [Acquiring data with Python](#) recipe in *Chapter 1, Using Python for Data Science*
- ▶ [Outliers](#) recipe in *Chapter 4, Analyzing Data - Deep Dive*

Grouping the data and using dot plots

EDA is about zooming in and out of the data from multiple angles in order to get a better grasp of the data. Let's now see the data from a different angle using dot plots. A dot plot is a simple plot where the data is grouped and plotted in a simple scale. It's up to us to decide how we want to group the data.



Dot plots are best used for small-sized to medium-sized datasets.
For large-sized data, a histogram is usually used.



Getting ready

For this exercise, we will use the same data as the previous section.

How to do it...

Let's load the necessary libraries. We will follow it up with the loading of our data and along the way, we will handle the missing values. Finally, we will group the data using a frequency counter:

```
# Load libraries
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from collections import OrderedDict
from matplotlib.pylab import frange

# 1.Load the data and handle missing values.
fill_data = lambda x : int(x.strip() or 0)
data = np.genfromtxt('president.txt', dtype=(int,int), converters={1:fill_data}, delimiter=",")
x = data[:,0]
y = data[:,1]

# 2.Group data using frequency (count of individual data points).
# Given a set of points, Counter() returns a dictionary, where key is
# a data point,
# and value is the frequency of data point in the dataset.
x_freq = Counter(y)
x_ = np.array(x_freq.keys()) y_ = np.array(x_freq.values())
```

We will proceed to group the data by the year range and plot it:

```
# 3. Group data by range of years
x_group = OrderedDict()
group= 5
group_count=1
keys = []
values = []
for i,xx in enumerate(x):
    # Individual data point is appended to list keys
    keys.append(xx)
    values.append(y[i])
    # If we have processed five data points (i.e. five years)
    if group_count == group:
        # Convert the list of keys to a tuple
        # use the new tuple as the key to x_group dictionary
        x_group[tuple(keys)] = values
        keys= []
        values =[]
        group_count = 1

    group_count+=1
# Accommodate the last batch of keys and values
x_group[tuple(keys)] = values

print x_group
# 4. Plot the grouped data as dot plot.
plt.subplot(311)
plt.title("Dot Plot by Frequency")
# Plot the frequency
plt.plot(y_,x_,'ro')
plt.xlabel('Count')
plt.ylabel('# Presedential Request')
# Set the min and max limits for x axis
plt.xlim(min(y_)-1,max(y_)+1)

plt.subplot(312)
plt.title("Simple dot plot")
plt.xlabel('# Presendtial Request')plt.ylabel('Frequency')
```

Finally, we will prepare the data for a simple dot plot and proceed with plotting it:

```
# Prepare the data for simple dot plot
# For every (item, frequency) pair create a
# new x and y
```

```
# where x is a list, created using np.repeat
# function, where the item is repeated frequency times.
# y is a list between 0.1 and frequency/10, incremented
# by 0.1
for key,value in x_freq.items():
    x__ = np.repeat(key,value)
    y__ = frange(0.1,(value/10.0),0.1)
    try:
        plt.plot(x__,y__,'go')
    except ValueError:
        print x__.shape, y__.shape
    # Set the min and max limits of x and y axis
    plt.ylim(0.0,0.4)
    plt.xlim(xmin=-1)

plt.xticks(x_freq.keys())

plt.subplot(313)
x_vals = []
x_labels = []
y_vals = []
x_tick = 1
for k,v in x_group.items():
    for i in range(len(k)):
        x_vals.append(x_tick)
        x_label = '-'.join([str(kk) if not i else str(kk)[-2:] for
i,kk in enumerate(k)])
        x_labels.append(x_label)
        y_vals.extend(list(v))
        x_tick+=1

plt.title("Dot Plot by Year Grouping")
plt.xlabel('Year Group')
plt.ylabel('No Presidential Request')
try:
    plt.plot(x_vals,y_vals,'ro')
except ValueError:
    print len(x_vals),len(y_vals)

plt.xticks(x_vals,x_labels,rotation=-35)plt.show()
```

How it works...

In step 1, we will load the data. This is the same as the data loading discussed in the previous recipe. Before we start plotting the data, we want to group them in order to see the overall data characteristics.

In steps 2 and 3, we will group the data using different criteria.

Let's look at step 2.

Here, we will use a function called `Counter()` from the `collections` package.



Given a set of points, `Counter()` returns a dictionary where key is a data point and value is the frequency of the data points in the dataset.



We will pass our dataset to `Counter()` and extract the keys from the actual data point and values, the respective frequency from this dictionary into numpy arrays `x_` and `y_` for ease of plotting. Thus, we have now grouped our data using frequency.

Before we move on to plot this, we will perform another grouping with this data in step 3.

We know that the x axis is years. Our data is also sorted by the year in an ascending order. In this step, we will group our data in a range of years, five in this case; that is, let's say that we will make a group from the first five years, our second group is the next five years, and so on:

```
group= 5
group_count=1
keys = []
values = []
```

The `group` variable defines how many years we want in a single group; in this example, we have 5 groups and `keys` and `values` are two empty lists. We will proceed to fill them with values from `x` and `y` till `group_count` reaches `group`, that is, 5:

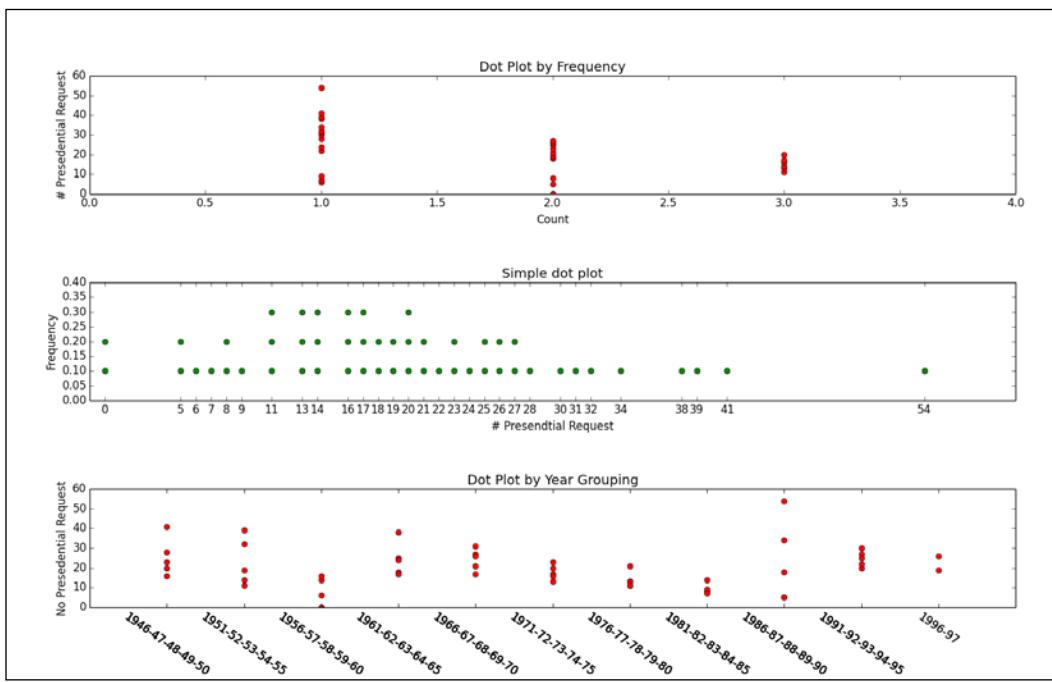
```
for i,xx in enumerate(x):
    keys.append(xx)
    values.append(y[i])
    if group_count == group:
        x_group[tuple(keys)] = values
        keys= []
        values =[]
        group_count = 0
        group_count+=1
        x_group[tuple(keys)] = values
```

The `x_group` is the name of the dictionary that now stores the group of values. We will need to preserve the order in which we will insert our records and so, we will use `OrderedDict` in this case.

[ OrderedDict preserves the order in which the keys are inserted.]

Now let's proceed to plot these values.

We want to plot all our graphs in a single window; hence, we will use the `subplot` parameter to the subplot, which defines the number of rows (3, the number in the hundredth place), number of columns (1, the number in the tenth place), and finally the plot number (1 in the unit place). Our plot output is as follows:



In the top graph, the data is grouped by frequency. Here, our x axis is the count and y axis is the number of Presidential Requests. We can see that 30 or more Presidential Requests have occurred only once. As said before, the dot plot is good at analyzing the range of the data points under different groupings.

The middle graph can be viewed as a very simple histogram. As the title of the graph (`in plt.title()`) says, it's the simplest form of a dot plot, where the x axis is the actual values and y axis is the number of times this x value occurs in the dataset. In a histogram, the bin size has to be set carefully; if not, it can distort the complete picture about the data. However, this can be avoided in this simple dot plot.

In the bottom graph, we have grouped the data by years.

See also

- ▶ *Creating Anonymous functions* recipe in *Chapter 1, Using Python for Data Science*
- ▶ *Pre-processing columns* recipe in *Chapter 1, Using Python for Data Science*
- ▶ *Acquiring data with Python* recipe in *Chapter 1, Using Python for Data Science*
- ▶ *Using Dictionary objects* recipe in *Chapter 1, Using Python for Data Science*

Using scatter plots for multivariate data

From a single column, we will now move on to multiple columns. In multivariate data analysis, we are interested in seeing if there are any relationships between the columns that we are analyzing. In two column/variable cases, the best place to start is a standard scatter plot. There can be four types of relationships, as follows:

- ▶ No relationship
- ▶ Strong
- ▶ Simple
- ▶ Multivariate (not simple) relationship

Getting ready

We will use the Iris dataset. It's a multivariate dataset introduced by Sir Ronald Fisher. Refer to <https://archive.ics.uci.edu/ml/datasets/Iris> for more information.

The Iris dataset has 150 instances and four attributes/columns. The 150 instances are composed of 50 records from each of the three species of the Iris flower (Setosa, virginica, and versicolor). The four attributes are the sepal length in cm, sepal width in cm, petal length in cm, and petal width in cm. Thus, the Iris dataset also serves as a great classification dataset. A classification method can be written in such a way that, given a record, we can classify which species that record belongs to after appropriate training.

How to do it...

Let's load the necessary libraries and extract the Iris data:

```
# Load Libraries
from sklearn.datasets import load_iris
import numpy as np
import matplotlib.pyplot as plt
import itertools

# 1. Load Iris dataset
data = load_iris()
x = data['data']
y = data['target']
col_names = data['feature_names']
```

We will proceed with demonstrating with a scatter plot:

```
# 2. Perform a simple scatter plot.
# Plot 6 graphs, combinations of our columns, sepal length, sepal
# width,
# petal length and petal width.
plt.close('all')
plt.figure(1)
# We want a plot with
# 3 rows and 2 columns, 3 and 2 in
# below variable signifies that.
subplot_start = 321
col_numbers = range(0,4)
# Need it for labeling the graph
col_pairs = itertools.combinations(col_numbers,2)
plt.subplots_adjust(wspace = 0.5)

for col_pair in col_pairs:
    plt.subplot(subplot_start)
    plt.scatter(x[:,col_pair[0]],x[:,col_pair[1]],c=y)
    plt.xlabel(col_names[col_pair[0]])
    plt.ylabel(col_names[col_pair[1]])
    subplot_start+=1plt.show()
```

How it works...

The scikit library provides a convenient function to load the Iris dataset called `load_iris()`. We will use this to load the Iris data in the variable `data` in step 1. The `data` is a dictionary object. Using the `data` and `target` keys, we will retrieve the records and class labels. We will look at the `x` and `y` values:

```
>>> x.shape  
(150, 4)  
>>> y.shape  
(150,)  
>>>
```

As you can see, `x` is a matrix with 150 rows and four columns; `y` is a vector of length 150. The data dictionary can also be queried to view the column names using the `feature_names` keyword, as follows:

```
>>> data['feature_names']  
  
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal  
width (cm)']  
>>>
```

We will then create a scatter plot of the iris variables in step 2. As we did before, we will use `subplot` here to accommodate all the plots in a single figure. We will get two combinations of our column using `itertools.combinations`:

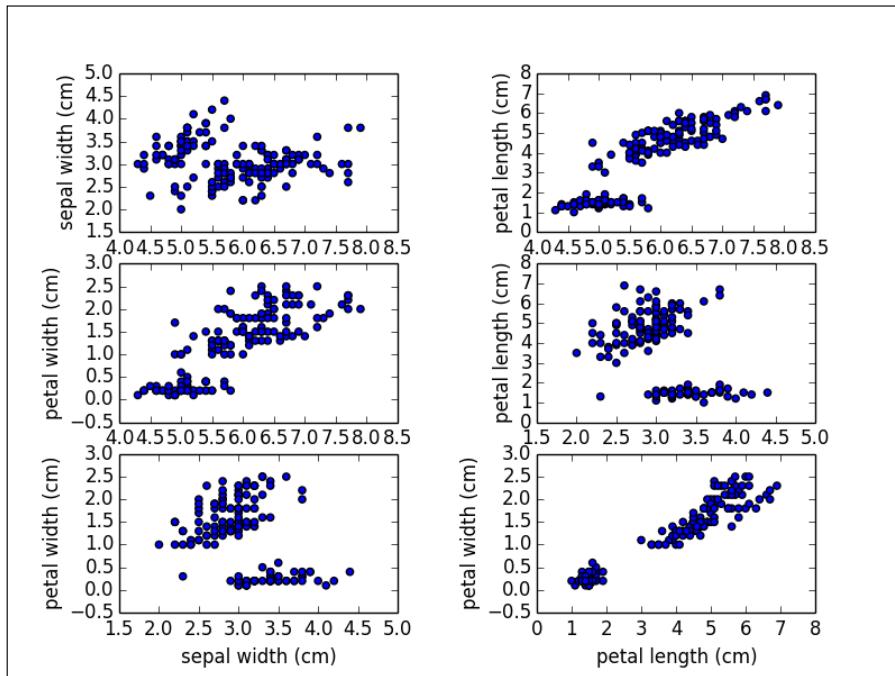
```
col_pairs = itertools.combinations(col_numbers, 2)
```

We can iterate `col_pairs` to get two combinations of our column and plot a scatter plot for each, as you can see in the following line of code:

```
plt.scatter(x[:, col_pair[0]], x[:, col_pair[1]], c=y)
```

We will pass a `c` parameter in order to indicate the color of the points. In this case, we will pass our `y` variable (class label) so that the different species of iris are plotted in different colors in our scatter plot.

The resulting plot is as follows:



As you can see, we have plotted two combinations of our columns. We also have the class labels represented using three different colors. Let's look at the bottom left plot, petal length versus petal width. We see that different range of values belong to different class labels. Now, this gives us a great clue for classification; the petal width and length variables are good candidates if the problem in hand is classification.

 For the Iris dataset, the petal width and length can alone classify the records in their respective flower family.

These kinds of observations can be quickly made during the feature selection process with the help of bivariate scatter plots.

See also

- ▶ [Using iterables recipe in Chapter 1, Using Python for Data Science](#)
- ▶ [Working with itertools recipe in Chapter 1, Using Python for Data Science](#)

Using heat maps

Heat maps are another interesting visualization technique. In a heat map, the data is represented as a matrix where the range of values taken by attributes are represented as color gradients. Look at the following Wikipedia reference for a general introduction to heat maps:

http://en.wikipedia.org/wiki/Heat_map

Getting ready

We will again resort to the Iris dataset in order to demonstrate how to build a heat map. We will also see the various ways that heat maps can be used on this data.

In this recipe, we will see how we can represent the whole data as a heat map and how the various interpretations of the data can be made from the heat map. Let's proceed to build a heat map of the Iris dataset.

How to do it...

Let's load the necessary libraries and import the Iris dataset. We will proceed with scaling the variables in the data by their mean value:

```
# Load libraries
from sklearn.datasets import load_iris
from sklearn.preprocessing import scale
import numpy as np
import matplotlib.pyplot as plt

# 1. Load iris dataset
data = load_iris()
x = data['data']
y = data['target']
col_names = data['feature_names']

# 2. Scale the variables, with mean value
x = scale(x,with_std=False)
x_ = x[1:26,]y_labels = range(1,26)
```

Let's plot our heat map:

```
# 3. Plot the Heat map
plt.close('all')

plt.figure(1)
```

```
fig,ax = plt.subplots()
ax.pcolor(x_,cmap=plt.cm.Greens,edgecolors='k')
ax.set_xticks(np.arange(0,x_.shape[1])+0.5)
ax.set_yticks(np.arange(0,x_.shape[0])+0.5)
ax.xaxis.tick_top()
ax.yaxis.tick_left()
ax.set_xticklabels(col_names,minor=False,fontsize=10)
ax.set_yticklabels(y_labels,minor=False,fontsize=10)plt.show()
```

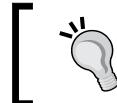
How it works...

In step 1, we will load the Iris dataset. Similar to the other recipes, we will take the data dictionary objects and store them as x and y for clarity. In step 2, we will scale the variables by their means:

```
x = scale(x,with_std=False)
```

With the parameter standard set to false, the scale function will use only the mean of the columns in order to normalize the data.

The reason for the scaling is to adjust the range of values that each column takes to a common scale, typically between 0 and 1. Having them in the same scale is very important for the heat map visualization as the values decide the color gradients.



Don't forget to scale your variables to bring them to the same range. Not having a proper scaling may lead to variables with a bigger range and scale, thus dominating others.



In step 3, we will perform the actual plotting. Before we plot, we will subset the data:

```
x = x[1:26,:]
col_names = data['feature_names']
y_labels = range(1,26)
```

As you can see, we selected only the first 25 records from the dataset. We did so in order to have the labels in the y axis to be readable. We will store the labels for the x and y axes in col_names and y_labels, respectively. Finally, we will use the pcolor function from pyplot to plot a heat map of the Iris data. We will do a little more tinkering with pcolor to make it look nice:

```
ax.set_xticks(np.arange(0,x.shape[1])+0.5)
ax.set_yticks(np.arange(0,x.shape[0])+0.5)
```

The x and y axis ticks are set uniformly:

```
ax.xaxis.tick_top()
```

Data Analysis – Explore and Wrangle _____

The x axis ticks are displayed at the top of the graph:

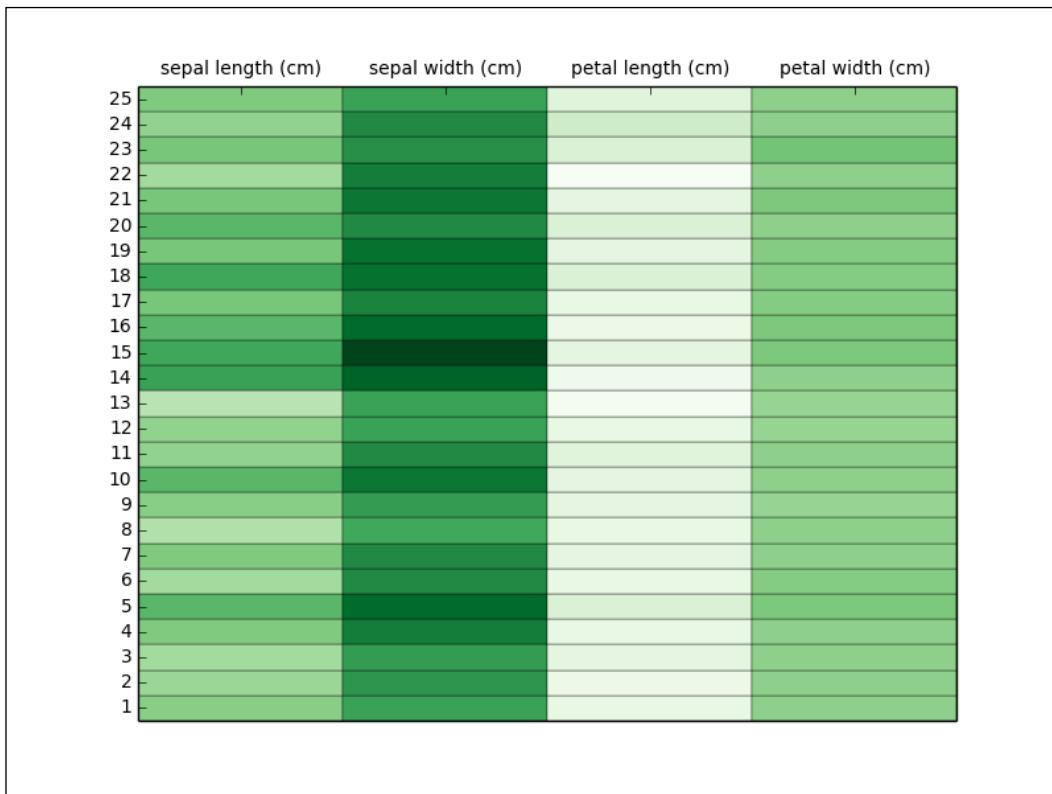
```
ax.xaxis.tick_left()
```

The y axis ticks are displayed to the left:

```
ax.set_xticklabels(col_names,minor=False,fontsize=10)  
ax.set_yticklabels(y_labels,minor=False,fontsize=10)
```

Finally, we will pass on the label values.

The output plot is shown as follows:



There's more...

Another interesting way to use a heat map is to view the variables separated by their respective classes; for example, in the Iris dataset, we will plot three different heat maps for the three classes that are present. The code is as follows:

```
x1 = x[0:50]
x2 = x[50:99]
x3 = x[100:149]

x1 = scale(x1,with_std=False)
x2 = scale(x2,with_std=False)
x3 = scale(x3,with_std=False)

plt.close('all')
plt.figure(2)
fig,(ax1, ax2, ax3) = plt.subplots(3, sharex=True, sharey=True)
y_labels = range(1,51)

ax1.set_xticks(np.arange(0,x.shape[1])+0.5)
ax1.set_yticks(np.arange(0,50,10))

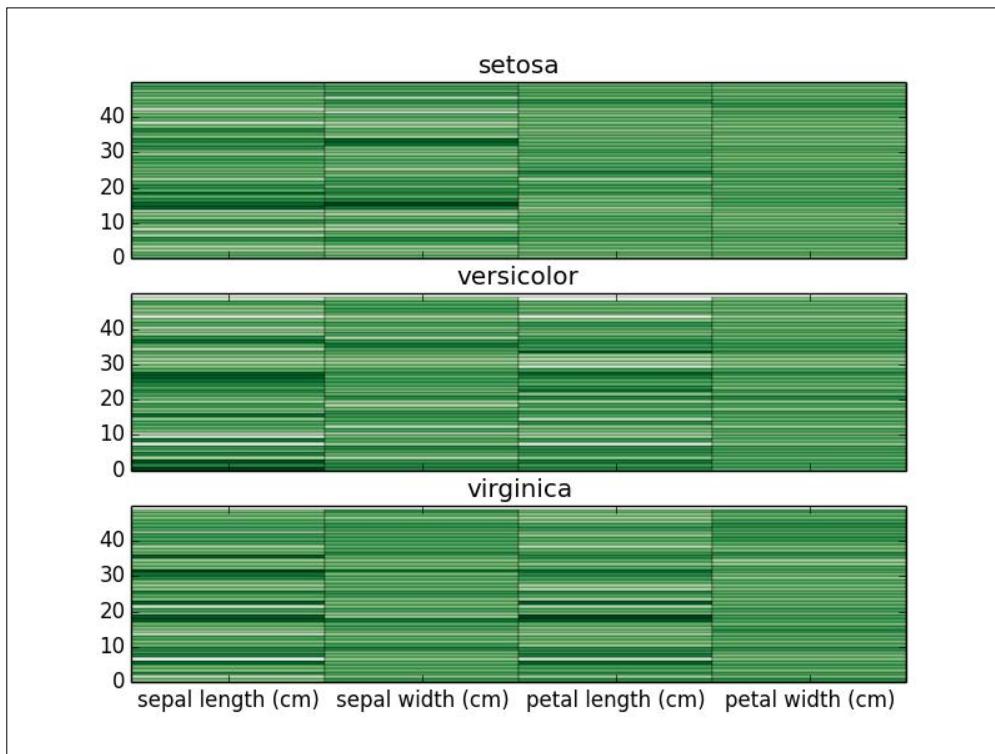
ax1.xaxis.tick_bottom()
ax1.set_xticklabels(col_names,minor=False,fontsize=2)

ax1.pcolor(x1,cmap=plt.cm.Greens,edgecolors='k')
ax1.set_title(data['target_names'][0])

ax2.pcolor(x2,cmap=plt.cm.Greens,edgecolors='k')
ax2.set_title(data['target_names'][1])

ax3.pcolor(x3,cmap=plt.cm.Greens,edgecolors='k')
ax3.set_title(data['target_names'][2])plt.show()
```

Let's look at the plot:



The first 50 records belong to the `setosa` class, the next 50 to `versicolor`, and the last 50 belong to `virginica`. We will make three heat maps for each of these classes.

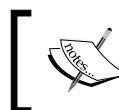
The cells are filled with the actual values of the records. You can notice that, for `setosa`, the sepal width has a good variation but doesn't show any significance in the case of `versicolor` and `virginica`.

See also

- ▶ Scaling the data recipe in Chapter 3, *Analyzing Data - Explore & Wrangle*

Performing summary statistics and plots

The primary purpose of using summary statistics is to get a good understanding of the location and dispersion of the data. By summary statistics, we refer to mean, median, and standard deviation. These quantities are quite easy to calculate. However, one should be careful when using them. If the underlying data is not unimodal, that is, it has multiple peaks, these quantities may not be of much use.



If the given data is unimodal, that is, having only one peak, the mean, which gives the location, and standard deviation, which gives the variance, are valuable metrics.

Getting ready

Let's use our Iris dataset to explore some of these summary statistics. In this section, we don't have a wholesome program producing a single output; however, we will have different steps demonstrating different summary measures.

How to do it...

Let's begin by importing the necessary libraries. We will follow it up with the loading of the Iris dataset:

```
# Load Libraries
from sklearn.datasets import load_iris
import numpy as np
from scipy.stats import trim_mean

# Load iris data
data = load_iris()
x = data['data']
y = data['target']
col_names = data['feature_names']
```

Let's now demonstrate how to calculate the mean, trimmed mean, and range values:

```
# 1.      Calculate and print the mean value of each column in the
Iris dataset
print "col name,mean value"
for i,col_name in enumerate(col_names):
    print "%s,%0.2f"%(col_name,np.mean(x[:,i]))
print

# 2.      Trimmed mean calculation.
```

```
p = 0.1 # 10% trimmed mean
print
print "col name,trimmed mean value"
for i,col_name in enumerate(col_names):
    print "%s,%0.2f"%(col_name,trim_mean(x[:,i],p))
print

# 3.      Data dispersion, calculating and display the range values.
print "col_names,max,min,range"
for i,col_name in enumerate(col_names):
    print "%s,%0.2f,%0.2f,%0.2f"%(col_name,max(x[:,i]),min(x[:,i]),max(x[:,i])-min(x[:,i]))
print
```

Finally, we will show the variance, standard deviation, mean absolute deviation, and median absolute deviation calculations:

```
# 4.      Data dispersion, variance and standard deviation
print "col_names,variance,std-dev"
for i,col_name in enumerate(col_names):
    print "%s,%0.2f,%0.2f"%(col_name,np.var(x[:,i]),np.std(x[:,i]))
print

# 5.      Mean absolute deviation calculation
def mad(x,axis=None):
    mean = np.mean(x,axis=axis)
    return np.sum(np.abs(x-mean))/(1.0 * len(x))

print "col_names,mad"
for i,col_name in enumerate(col_names):
    print "%s,%0.2f"%(col_name,mad(x[:,i]))
print

# 6.      Median absolute deviation calculation
def mdad(x,axis=None):
    median = np.median(x,axis=axis)
    return np.median(np.abs(x-median))

print "col_names,median,median abs dev,inter quartile range"
for i,col_name in enumerate(col_names):
    iqr = np.percentile(x[:,i],75) - np.percentile(x[:,i],25)
    print "%s,%0.2f,%0.2f,%0.2f"%(col_name,np.median(x[:,i]),
        mdad(x[:,i]),iqr)
print
```

How it works...

The loading of the Iris dataset is not repeated in this recipe. It's assumed that the reader can look at the previous recipe to do the same. Further, we will assume that the `x` variable is loaded with all the instance of the Iris records with each record having four columns.

Step 1 prints the mean value of each of the column in the Iris dataset. We used NumPy's `mean` function for the same. The output of the print statement is as follows:

```
col name,mean value
sepal length (cm),5.84
sepal width (cm),3.05
petal length (cm),3.76
petal width (cm),1.20
```

As you can see, we have the mean value for each column. The code to calculate the mean is as follows:

```
np.mean(x[:, i])
```

We passed all the rows and columns in the loop. Thus, we get the mean value by columns.

Another interesting measure is what is called trimmed mean. It has its own advantages. The 10% trimmed mean of a given sample is computed by excluding the 10% largest and 10% smallest values from the sample and taking the arithmetic mean of the remaining 80% of the sample.



Compared to the regular mean, a trimmed mean is less sensitive to outliers.



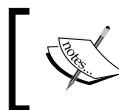
SciPy provides us with a `trim_mean` function. We will demonstrate the trimmed mean calculation in step 2. The output is as follows:

```
col name,trimmed mean value
sepal length (cm),5.81
sepal width (cm),3.04
petal length (cm),3.76
petal width (cm),1.18
```

With the Iris dataset, we don't see a lot of difference, but in real-world datasets, the trimmed mean is very handy as it gives a better picture of the location of the data.

Till now, what we saw was the location of the data and that the mean and trimmed mean gives a good inference on the data location. Another important aspect to look at is the dispersion of the data. The simplest way to look at the data dispersion is range, which is defined as follows, given a set of values, x , the range is the maximum value of x – minimum value of x . In Step 3, we will calculate and print the same:

```
col_names,max,min,range
sepal length (cm),7.90,4.30,3.60
sepal width (cm),4.40,2.00,2.40
petal length (cm),6.90,1.00,5.90
petal width (cm),2.50,0.10,2.40
```



If the data falls in a very narrow range, say, most of the values cluster around a single value and we have a few extreme values, then the range may be misleading.



When the data falls in a very narrow range and clusters around a single value, variance is used as a typical measure of the dispersion/spread of the data. Variance is the sum of the squared difference between the individual values and the mean value divided by the number of instances. In step 4, we will see the variance calculation.

In the preceding code, in addition to variance, we can see std-dev, that is, standard deviation. As variance is the square of the difference, it's not in the same measurement scale as the original data. We will use standard deviation, which is the square root of the variance, in order to get the data back into its original scale. Let's look at the output of the print statement, where we listed both the variance and standard deviation:

```
col_names,variance,std-dev
sepal length (cm),0.68,0.83
sepal width (cm),0.19,0.43
petal length (cm),3.09,1.76
petal width (cm),0.58,0.76
```

As we mentioned earlier, the mean is very sensitive to outliers; variance also uses the mean, and hence, it's prone to the same issues as the mean. We can use other measures for variance to avoid this trap. One such measure is absolute average deviation; instead of taking the square of the difference between the individual values and mean and dividing it by the number of instances, we will take the absolute of the difference between the mean and individual values and divide it by the number of instances. In step 5, we will define a function for this:

```
def mad(x, axis=None):
    mean = np.mean(x, axis=axis)
    return np.sum(np.abs(x-mean)) / (1.0 * len(x))
```

As you can see, the function returns the absolute difference between the mean and individual values. The output of this step is as follows:

```
col_names,mad
sepal length (cm),0.69
sepal width (cm),0.33
petal length (cm),1.56
petal width (cm),0.66
```

With the data having many outliers, there is another set of metrics that come in handy. They are the median and percentiles. We already saw percentiles in the previous section while plotting the univariate data. Traditionally, median is defined as a value from the dataset such that half of all the points in the dataset are smaller and the other half is larger than the median value.



Percentiles are a generalization of the concept of median. The 50th percentile is the traditional median value.



We saw the 25th and 75th percentiles in the previous section. The 25th percentile is a value such that 25% of all the points in the dataset are smaller than this value:

```
>>>
>>> a = [8,9,10,11]
>>> np.median(a)
9.5
>>> np.percentile(a,50)
9.5
```

The median is the measure of the location of the data distribution. Using percentiles, we can get a metric for the dispersion of the data, the interquartile range. The interquartile range is the distance between the 75th percentile and 25th percentile. Similar to the mean absolute deviation as explained previously, we also have the median absolute deviation.

In step 6, we will calculate and display both the interquartile range and median absolute deviation. We will define the following function in order to calculate the median absolute deviation:

```
def mdad(x, axis=None):
    median = np.median(x, axis=axis)
    return np.median(np.abs(x-median))
```

The output is as follows:

```
col_names,median,median abs dev,inter quartile range
sepal length (cm),5.80,0.70,5.30
sepal width (cm),3.00,0.25,2.20
petal length (cm),4.35,1.25,4.07
petal width (cm),1.30,0.70,0.62
```

See also

- ▶ *Grouping Data and Using Plots* recipe in Chapter 3, *Analyzing Data - Explore & Wrangle*

Using a box-and-whisker plot

A box-and-whisker plot is a good companion with the summary statistics to view the statistical summary of the data in hand. Box-and-whiskers can effectively represent quantiles in data and also outliers, if any, emphasizing the overall structure of the data. A box plot consists of the following features:

- ▶ A horizontal line indicating the median that indicates the location of the data
- ▶ A box spanning the interquartile range, measuring the dispersion
- ▶ A set of whiskers that extends from the central box horizontally and vertically, which indicates the tail of the distribution

Getting ready

Let's use the box plot to look at the Iris dataset.

How to do it...

Let's load the necessary libraries to begin with. We will follow this with loading the Iris dataset:

```
# Load Libraries
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

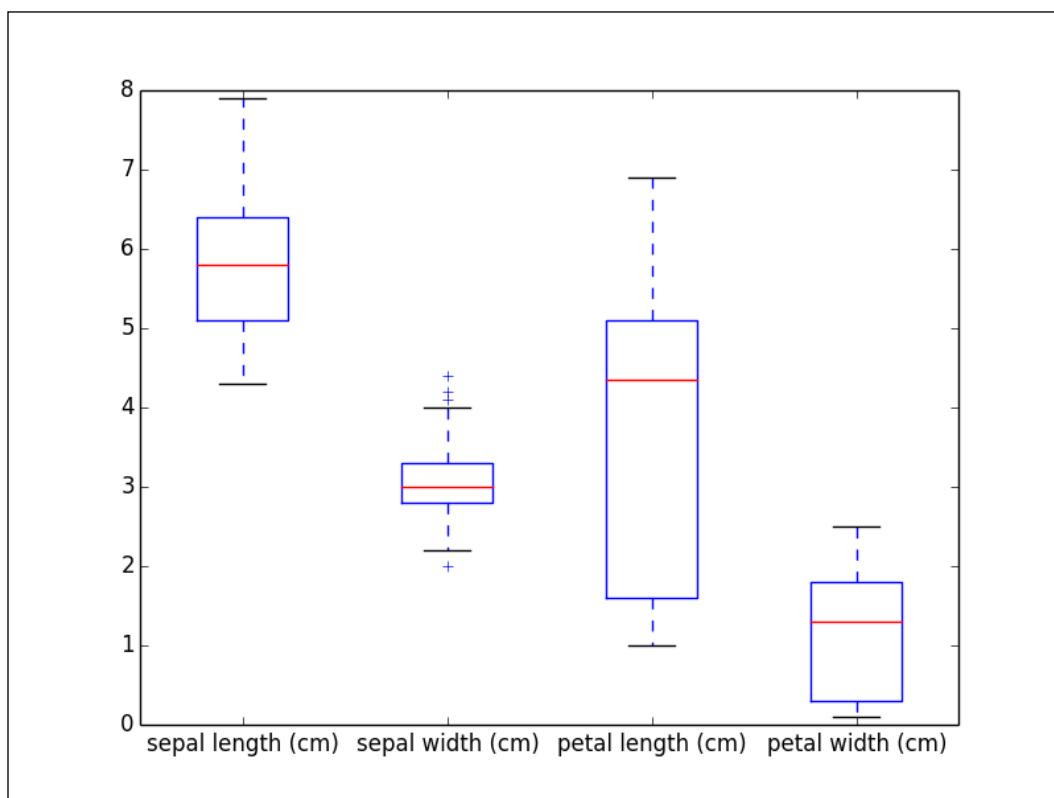
# Load Iris dataset
data = load_iris()
x = data['data']
plt.close('all')
```

Let's demonstrate how to create a box-and-whisker plot:

```
# Plot the box and whisker
fig = plt.figure(1)
ax = fig.add_subplot(111)
ax.boxplot(x)
ax.set_xticklabels(data['feature_names'])
plt.show()
```

How it works...

The code is very straightforward. We will load the Iris data in `x` and pass the `x` values to the box plot function from pyplot. As you know, our `x` has four columns. The box plot is as follows:



The box plot has captured both the location and variation of all the four columns in a single plot.

The horizontal red line indicates the median, which is the location of the data. You can see that the sepal length has a higher median than the rest of the columns.

The box spanning the interquartile range measuring the dispersion can be seen for all the four variables.

You can see a set of whiskers that extends from the central box horizontally and vertically, which indicates the tail of the distribution. Whiskers help you to see the extreme values in the datasets.

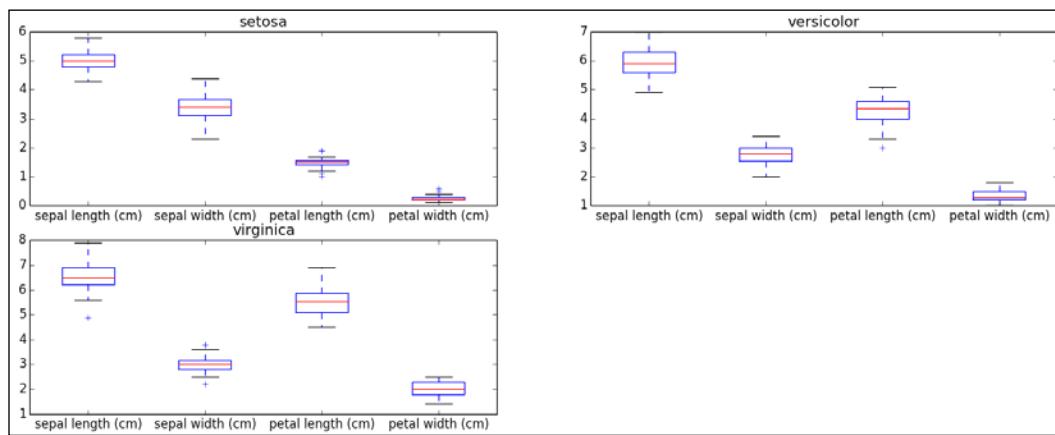
There's more...

It will also be interesting to see how the data is distributed across the various class labels. Similar to how we did in the scatter plots, let's do the same with the box-and-whisker plot. The following code and chart explains how to plot a box plot across various class labels:

```
y=data['target']
class_labels = data['target_names']

fig = plt.figure(2,figsize=(18,10))
sub_plt_count = 321
for t in range(0,3):
    ax = fig.add_subplot(sub_plt_count)
    y_index = np.where(y==t)[0]
    x_ = x[y_index,:]
    ax.boxplot(x_)
    ax.set_title(class_labels[t])
    ax.set_xticklabels(data['feature_names'])
    sub_plt_count+=1
plt.show()
```

As you can see in the following chart, we now have a box-and-whisker plot for each class label:



Imputing the data

In many real-world scenarios, we have the problem of incomplete or missing data. We need a strategy to handle the incomplete data. This strategy can be formulated either using the data alone or in conjunction with the class labels, if the labels are present.

Getting ready

Let's first look at the ways of imputing the data without using the class labels.

A simple technique is to ignore the missing value and hence, avoid the overhead of data imputation. However, this can be applied when the data is available in abundance, which is not always the case. If the dataset has very few missing values and the percentage of the missing values is minimal, we can ignore them. Typically, it's not about ignoring a single value of a variable, it's about ignoring a tuple that contains this variable. We have to be more careful when ignoring a whole tuple, as the other attributes in this tuple may be very critical for our task.

A better way to handle the missing data is to estimate it. Now, the estimation process can be carried out considering only the data or in conjunction with the class label. In the case of a continuous variable, the mean, median, or the most frequent value can be used to replace the missing value. Scikit-learn provides you with an `Imputer()` function in module preprocessing to handle the missing data. Let's see an example where we will perform data imputation. To better understand the imputation technique, we will artificially introduce some missing values in the Iris dataset.

How to do it...

Let's load the necessary libraries to begin with. We will load the Iris dataset as usual and introduce some arbitrary missing values:

```
# Load Libraries
from sklearn.datasets import load_iris
from sklearn.preprocessing import Imputer
import numpy as np
import numpy.ma as ma

# 1. Load Iris Data Set
data = load_iris()
x = data['data']
y = data['target']

# Make a copy of the original x value
```

```
x_t = x.copy()

# 2.      Introduce missing values into second row
x_t[2,:] = np.repeat(0,x.shape[1])
```

Let's see some data imputation in action:

```
# 3.      Now create an imputer object with strategy as mean,
# i.e. fill the missing values with the mean value of the missing
# column.
imputer = Imputer(missing_values=0,strategy="mean")
x_imputed = imputer.fit_transform(x_t)

mask = np.zeros_like(x_t)
mask[2,:] = 1
x_t_m = ma.masked_array(x_t,mask)

print np.mean(x_t_m, axis=0) print x_imputed[2,:]
```

How it works...

Step 1 is about loading the Iris data in memory. In step 2, we will introduce some missing values; in this case, we will set all the columns in the third row to 0.

In step 3, we will use the Imputer object to handle the missing data:

```
imputer = Imputer(missing_values=0,strategy="mean")
```

As you can see, we will need two parameters, `missing_values` to specify the missing values, and `strategy`, which is a way to impute these missing values. The Imputer object provides the following three strategies:

- ▶ `mean`
- ▶ `median`
- ▶ `most_frequent`

Using the `mean`, any cell with the 0 value will be replaced by the mean value of the column that the cell belongs to. In the case of the `median`, the median value is used to replace 0, and in `most_frequent`, as the name suggests, the most frequent value is used to replace 0. Based on the context of our application, one of these strategies can be applied.

The initial value of `x[2,:]` is as follows:

```
>>> x[2,:]
array([ 4.7,  3.2,  1.3,  0.2])
```

We will make it 0 in all the columns and use an imputer with the mean strategy.

Before we look at the imputer output, let's calculate the mean values for all the columns:

```
import numpy.ma as ma
mask = np.zeros_like(x_t)
mask[2, :] = 1
x_t_m = ma.masked_array(x_t, mask)

print np.mean(x_t_m, axis=0)
```

The output is as follows:

```
[5.851006711409397 3.053020134228189 3.7751677852349017
1.2053691275167793]
```

Now, let's look at the imputed output for row number 2:

```
print x_imputed[2, :]
```

The following is the output:

```
[ 5.85100671  3.05302013  3.77516779  1.20536913]
```

As you can see, the imputer has filled the missing values with the mean value of the respective columns.

There's more...

As we discussed, we can also leverage the class labels and impute the missing values either using the mean or median:

```
# Impute based on class label
missing_y = y[2]
x_missing = np.where(y==missing_y) [0]
y = data['target']
# Mean stragegy
print np.mean(x[x_missing, :], axis=0)
# Median stragegy
print np.median(x[x_missing, :], axis=0)
```

Instead of using the mean or median of the whole dataset, what we did was to subset the data by the class variable of the missing tuple:

```
missing_y = y[2]
```

We introduced the missing value in the third record. We will take the class label associated with this record to the `missing_y` variable:

```
x_missing = np.where(y==missing_y) [0]
```

Now, we will take all the tuples that have the same class label:

```
# Mean stragegy  
print np.mean(x[x_missing,:],axis=0)  
# Median stragegy  
print np.median(x[x_missing,:],axis=0)
```

We can now apply the mean or median strategy by replacing the missing tuple with the mean or median of all the tuples that belong to this class label.

We took the mean/median value of this subset for the data imputation process.

See also

- ▶ *Performing Summary Statistics* recipe in *Chapter 3, Analyzing Data - Explore & Wrangle*

Performing random sampling

In this we will learn to how to perform a random sampling of data.

Getting ready

Typically, in scenarios where it's very expensive to access the whole dataset, sampling can be effectively used to extract a portion of the dataset for analysis. Sampling can be effectively used in EDA as well. A sample should be a good representative of the underlying dataset. It should have approximately the same characteristics as the underlying dataset. For example, with respect to the mean, the sample mean should be as close to the original data's mean value as possible. There are several sampling techniques; we will cover one of them here.

In simple random sampling, there is an equal chance of selecting any tuple. For our example, we want to sample ten records randomly from the Iris dataset.

How to do it...

We will begin with loading the necessary libraries and importing the Iris dataset:

```
# Load libraries  
from sklearn.datasets import load_iris
```

```
import numpy as np

# 1.      Load the Iris data set
data = load_iris()
x = data['data']
```

Let's demonstrate how sampling is performed:

```
# 2.      Randomly sample 10 records from the loaded dataset
no_records = 10
x_sample_indx = np.random.choice(range(x.shape[0]), no_records)
print x[x_sample_indx, :]
```

How it works...

In step 1, we will load the Iris dataset. In step 2, we will do a random selection using the `choice` function from `numpy.random`.

The two parameters that we will pass to the `choice` functions are a range variable for the total number of rows in the original dataset and the sample size that we require. From zero to the total number of rows in the original dataset, the `choice` function randomly picks n integers, where n is the size of the sample, which is dictated by `no_records` in our case.

Another important aspect is that one of the parameters to the `choice` function is `replace` and it's set to True by default; it specifies whether we need to sample with replacement or without replacement. Sampling without replacement removes the sampled item from the original list so it will not be a candidate for future sampling. Sampling with replacement does the opposite; every element has an equal chance to be sampled in future sampling even though it's been sampled before.

There's more...

Stratified sampling

If the underlying dataset consists of different groups, a simple random sampling may fail to capture adequate samples in order to be able to represent the data. For example, in a two-class classification problem, 10% of the data belongs to the positive class and 90% belongs to the negative class. This kind of problem is called class imbalance problem in machine learning. When we do sampling on such imbalanced datasets, the sample should also reflect the preceding percentages. This kind of sampling is called stratified sampling. We will look more into stratified sampling in future chapters on machine learning.

Progressive sampling

How do we determine the correct sample size that we need for a given problem? We discussed several sampling techniques before but we don't have a strategy to select the correct sample size. There is no simple answer for this. One way to do this is to use progressive sampling. Select a sample size and get the samples through any of the sampling techniques, apply the desired operation on the data, and record the results. Now, increase the sample size and repeat the steps. This iterative process is called progressive sampling.

Scaling the data

In this we will learn to how to scale the data.

Getting ready

Scaling is an important type of data transformation. Typically, by doing scaling on a dataset, we can control the range of values that the data type can assume. In a dataset with multiple columns, the columns with a bigger range and scale tend to dominate other columns. We will perform scaling of the dataset in order to avoid these interferences.

Let's say that we are comparing two software products based on the number of features and the number of lines of code. The difference in the number of lines of code will be very high compared to the difference in the number of features. In this case, our comparison will be dominated by the number of lines of code. If we use any similarity measure, the similarity or difference will be dominated by the number of lines of code. To avoid such a situation, we will adopt scaling. The simplest scaling is min-max scaling. Let's look at min-max scaling on a randomly generated dataset.

How to do it...

Let's generate some random data in order to test our scaling functionality:

```
# Load Libraries
import numpy as np

# 1.      Generate some random data for scaling
np.random.seed(10)
x = [np.random.randint(10,25)*1.0 for i in range(10)]
```

Now, we will demonstrate scaling:

```
# 2.Define a function, which can perform min max scaling given a list
of numbers
def min_max(x):
```

```
return [round((xx-min(x))/(1.0*(max(x)-min(x))),2) for xx in x]

# 3.Perform scaling on the given input list.
print x
print min_max(x)
```

How it works...

In step 1, we will generate a list of random numbers between 10 and 25. In step 2, we will define a function to perform min-max scaling on the given input. Min-max scaling is defined as follows:

```
x_scaled = x - min(x) / max(x) -min (x)
```

In step 2 we define a function to do the above task.

This transforms the range of the given value. After transformation, the values will fall in the [0,1] range.

In step 3, we will first print the original input list. The output is as follows:

```
[19, 23, 14, 10, 11, 21, 22, 19, 23, 10]
```

We will pass this list to our `min_max` function in order to get the scaled output, which is as follows:

```
[0.69, 1.0, 0.31, 0.0, 0.08, 0.85, 0.92, 0.69, 1.0, 0.0]
```

You can see the scaling in action; 10, which is the smallest number, has been assigned a value of 0 . 0 and 23, the highest number, is assigned a value of 1 . 0. Thus, we scaled the data in the [0,1] range.

There's more...

Scikit-learn provides a `MinMaxScaler` function for the same:

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np

np.random.seed(10)
x = np.matrix([np.random.randint(10,25)*1.0 for i in range(10)])
x = x.T
minmax = MinMaxScaler(feature_range=(0.0,1.0))
print x
x_t = minmax.fit_transform(x)
print x_t
```

The output is as follows:

```
[19.0, 23.0, 14.0, 10.0, 11.0, 21.0, 22.0, 19.0, 23.0, 10.0]
```

```
[0.69, 1.0, 0.31, 0.0, 0.08, 0.85, 0.92, 0.69, 1.0, 0.0]
```

We saw examples where we scaled the data to a range (0,1); this can be extended to any range. Let's say that our new range is `nr_min, nr_max`, then the min-max formula is modified as follows:

```
x_scaled = (x - min(x) / max(x) -min (x) ) * (nr_max- nr_min) + nr_min
```

The following will be the Python code:

```
import numpy as np

np.random.seed(10)
x = [np.random.randint(10,25)*1.0 for i in range(10)]

def min_max_range(x, range_values):
    return [round( ((xx-min(x))/(1.0*(max(x)-min(x))))*(range_values[1]-range_values[0]) \
    + range_values[0],2) for xx in x]

print min_max_range(x, (100,200))
```

where, `range_values` is a tuple of two elements, where the 0th element is the new range's lower end and the first element is the higher end. Let's invoke this function on our input and see how the output is, as follows:

```
print min_max_range(x, (100,200))
```

```
[169.23, 200.0, 130.77, 100.0, 107.69, 184.62, 192.31, 169.23, 200.0,
100.0]
```

The lowest value, 10, is now scaled to 100 and the highest value, 23, is scaled to 200.

Standardizing the data

Standardization is the process of converting the input so that it has a mean of 0 and standard deviation of 1.

Getting ready

If you are given a vector X, the mean of 0 and standard deviation of 1 for X can be achieved by the following equation:


$$\text{Standardized } X = \frac{x - \text{mean}(value)}{\text{standard deviation}(X)}$$

Let's see how this can be achieved in Python.

How to do it...

Let's import the necessary libraries to begin with. We will follow this with the generation of the input data:

```
# Load Libraries
import numpy as np
from sklearn.preprocessing import scale

# Input data generation
np.random.seed(10)
x = [np.random.randint(10, 25)*1.0 for i in range(10)]
```

We are now ready to demonstrate standardization:

```
x_centered = scale(x, with_mean=True, with_std=False)
x_standard = scale(x, with_mean=True, with_std=True)

print x
print x_centered
print x_standard
print "Orginal x mean = %0.2f, Centered x mean = %0.2f, Std dev of \
      standard x =%0.2f"%(np.mean(x), np.mean(x_centered), np.std(x_
      standard))
```

How it works...

We will generate some random data using np.random:

```
x = [np.random.randint(10, 25)*1.0 for i in range(10)]
```

We will perform standardization using the `scale` function from scikit-learn:

```
x_centered = scale(x, with_mean=True, with_std=False)
x_standard = scale(x, with_mean=True, with_std=True)
```

The `x_centered` is scaled using only the mean; you can see the `with_mean` parameter set to `True` and `with_std` set to `False`.

The `x_standard` is standardized using both mean and standard deviation.

Now let us look at the output.

The original data is as follows:

```
[19.0, 23.0, 14.0, 10.0, 11.0, 21.0, 22.0, 19.0, 23.0, 10.0]
```

Next, we will print `x_centered`, where we centered it with the mean value:

```
[ 1.8  5.8 -3.2 -7.2 -6.2  3.8  4.8  1.8  5.8 -7.2]
```

Finally we will print `x_standardized`, where we used both the mean and standard deviation:

```
[ 0.35059022  1.12967961 -0.62327151 -1.4023609 -1.20758855
 0.74013492
 0.93490726  0.35059022  1.12967961 -1.4023609 ]
```

```
Orginal x mean = 17.20, Centered x mean = 0.00, Std dev of standard x
=1.00
```

There's more...



Standardization can be generalized to any level and spread, as follows:

Standardized value = value – level / spread



Let's break the preceding equation in two parts: just the numerator part, which is called centering, and the whole equation, which is called standardization. Using the mean values, centering plays a critical role in regression. Consider a dataset that has two attributes, weight and height. We will center the data such that the predictor, weight, has a mean of 0. This makes the interpretation of intercept easier. The intercept will be interpreted as what is the expected height when the predictor values are set to their mean.

Performing tokenization

When you are given any text, the first job is to tokenize the text into a format that is based on the given problem requirements. Tokenization is a very broad term; we can tokenize the text at the following various levels of granularity:

- ▶ The paragraph level
- ▶ The sentence level
- ▶ The word level

In this section, we will see sentence level and word level tokenization. The methods are similar and can be easily applied to a paragraph level or any other level of granularity as required by the problem at hand.

Getting ready

We will see how to perform sentence level and word level tokenization in a single recipe.

How to do it...

Let's start with the demonstration of sentence tokenization:

```
# Load Libraries
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
from collections import defaultdict

# 1.Let us use a very simple text to demonstrate tokenization
# at sentence level and word level. You have seen this example in the
# dictionary recipe, except for some punctuation which are added.

sentence = "Peter Piper picked a peck of pickled peppers. A peck of
pickled \
peppers, Peter Piper picked !!! If Peter Piper picked a peck of
pickled \
peppers, Wheres the peck of pickled peppers Peter Piper picked ?"

# 2.Using nltk sentence tokenizer, we tokenize the given text into
sentences
```

```
# and verify the output using some print statements.

sent_list = sent_tokenize(sentence)

print "No sentences = %d"%(len(sent_list))
print "Sentences"
for sent in sent_list: print sent

# 3. With the sentences extracted let us proceed to extract
# words from these sentences.
word_dict = defaultdict(list)
for i,sent in enumerate(sent_list):
    word_dict[i].extend(word_tokenize(sent))

print word_dict
```

A quick peek at how NLTK performs its sentence tokenization in the following way:

```
def sent_tokenize(text, language='english'):
    """
    Return a sentence-tokenized copy of *text*,
    using NLTK's recommended sentence tokenizer
    (currently :class:`.PunktSentenceTokenizer`
    for the specified language).

    :param text: text to split into sentences
    :param language: the model name in the Punkt corpus
    """
    tokenizer = load('tokenizers/punkt/{0}.pickle'.format(language))
    return tokenizer.tokenize(text)
```

How it works...

In step 1, we will initialize a variable sentence with a paragraph. This is the same example that we used in the dictionary recipe. In step 2, we will use nltk's sent_tokenize function to extract sentences from the given text. You can look into the source of sent_tokenize in nltk in the documentation found at http://www.nltk.org/api/nltk.tokenize.html#nltk.tokenize.sent_tokenize.

As you can see, sent_tokenize loads a prebuilt tokenizer model, and using this model, it tokenizes the given text and returns the output. The tokenizer model is an instance of PunktSentenceTokenizer from the nltk.tokenize.punkt module. There are several pretrained instances of this tokenizer available in different languages. In our case, you can see that the language parameter is set to English.

Let's look at the output of this step:

```
No sentences = 3
Sentences
Peter Piper picked a peck of pickled peppers.
A peck of pickled           peppers, Peter Piper picked !!!
If Peter Piper picked a peck of pickled           peppers, Wheres
the peck of pickled peppers Peter Piper picked ?
```

As you can see, the sentence tokenizer has split our input text into three sentences. Let's proceed to step 3, where we will tokenize these sentences into words. Here, we will use the `word_tokenize` function in order to extract the words from each of the sentences and store them in a dictionary, where the key is the sentence number and the value is the list of words for that sentence. Let's look at the output of the print statement:

```
defaultdict(<type 'list'>, {0: ['Peter', 'Piper', 'picked', 'a',
'peck', 'of', 'pickled', 'peppers', '.'], 1: ['A', 'peck', 'of',
'pickled', 'peppers', ',', 'Peter', 'Piper', 'picked', '!', '!', '!'],
2: ['If', 'Peter', 'Piper', 'picked', 'a', 'peck', 'of', 'pickled',
'peppers', ',', 'Wheres', 'the', 'peck', 'of', 'pickled', 'peppers',
'Peter', 'Piper', 'picked', '?']})
```

The `word_tokenize` uses a regular expression to split the sentences into words. It will be useful to look at the source of `word_tokenize` found at http://www.nltk.org/_modules/nltk/tokenize/punkt.html#PunktLanguageVars.word_tokenize.

There's more...

For sentence tokenization, we saw a way of doing it in NLTK. There are other methods available. The `nltk.tokenize.simple` module has a `line_tokenize` method. Let's take the same input sentence as before and run it using `line_tokenize`:

```
# Load Libraries
from nltk.tokenize import line_tokenize

sentence = "Peter Piper picked a peck of pickled peppers. A peck of
pickled \
peppers, Peter Piper picked !!! If Peter Piper picked a peck of
pickled \
peppers, Wheres the peck of pickled peppers Peter Piper picked ?"

sent_list = line_tokenize(sentence)
print "No sentences = %d"%(len(sent_list))
print "Sentences"
```

```
for sent in sent_list: print sent

# Include new line characters
sentence = "Peter Piper picked a peck of pickled peppers. A peck of
pickled\n \
peppers, Peter Piper picked !!! If Peter Piper picked a peck of
pickled\n \
peppers, Wheres the peck of pickled peppers Peter Piper picked ?"

sent_list = line_tokenize(sentence)
print "No sentences = %d"%(len(sent_list))
print "Sentences"
for sent in sent_list: print sent
```

The output is as follows:

```
No sentences = 1
Sentences
Peter Piper picked a peck of pickled peppers. A peck of pickled
peppers, Peter Piper picked !!! If Peter Piper picked a peck of
pickled           peppers, Wheres the peck of pickled peppers Peter
Piper picked ?
```

You can see that we have only the sentence retrieved from the input.

Let's now modify our input in order to include new line characters:

```
sentence = "Peter Piper picked a peck of pickled peppers. A peck of
pickled\n \
peppers, Peter Piper picked !!! If Peter Piper picked a peck of
pickled\n \
peppers, Wheres the peck of pickled peppers Peter Piper picked ?"
```

Note that we have a new line character added. We will again apply `line_tokenize` to get the following output:

```
No sentences = 3
Sentences
Peter Piper picked a peck of pickled peppers. A peck of pickled
          peppers, Peter Piper picked !!! If Peter Piper picked a
peck of pickled
          peppers, Wheres the peck of pickled peppers Peter Piper
picked ?
```

You can see that it has tokenized our sentences at the new line and now we have three sentences.

See Chapter 3 of the *NLTk* book; it has more references for sentence and word tokenization. It can be found at <http://www.nltk.org/book/ch03.html>.

See also

- ▶ [Using Dictionary object recipe in Chapter 1, Using Python for Data Science](#)
- ▶ [Writing list recipe in Chapter 1, Using Python for Data Science](#)

Removing stop words

In text processing, we are interested in words or phrases that will help us differentiate the given text from the other text in the corpus. Let's call these words or phrases as key phrases. Every text mining application needs a way to find out the key phrases. An information retrieval application needs key phrases for the easy retrieval and ranking of search results. A text classification system needs key phrases as its features that are to be fed to a classifier.

This is where stop words come into the picture.

"Sometimes, some extremely common words which would appear to be of little value in helping select documents matching a user need are excluded from the vocabulary entirely. These words are called stop words."

Introduction to Information Retrieval By Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze

The Python NLTK library provides us with a default stop word corpus that we can leverage, as follows:

```
>>> from nltk.corpus import stopwords  
>>> stopwords.words('english')  
[u'i', u'me', u'my', u'myself', u'we', u'our', u'ours', u'ourselves',  
u'you', u'your', u'yours', u'yourself', u'yourselves', u'he',  
u'him', u'his', u'himself', u'she', u'her', u' hers', u'herself',  
u'it', u'its', u'itself', u'they', u'them', u'their', u'theirs',  
u'themselves', u'what', u'which', u'who', u'whom', u'this', u'that',  
u'these', u'those', u'am', u'is', u'are', u'was', u'were', u'be',  
u'been', u'being', u'have', u'has', u'had', u'having', u'do', u'does',  
u'did', u'doing', u'a', u'an', u'the', u'and', u'but', u'if', u'or',  
u'because', u'as', u'until', u'while', u'of', u'at', u'by', u'for',  
u'with', u'about', u'against', u'between', u'into', u'through',  
u'during', u'before', u'after', u'above', u'below', u'to', u'from',  
u'up', u'down', u'in', u'out', u'on', u'off', u'over', u'under',  
u'again', u'further', u'then', u'once', u'here', u'there', u'when',  
u'where', u'why', u'how', u'all', u'any', u'both', u'each', u'few',  
u'more', u'most', u'other', u'some', u'such', u'no', u'nor', u'not',  
u'only', u'own', u'same', u'so', u'than', u'too', u'very', u's', u't',  
u'can', u'will', u'just', u'don', u'should', u'now']  
>>>
```

You can see that we have printed the list of stop words in English.

How to do it...

Let's load the necessary library and introduce our input text:

```
# Load libraries
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import string

text = "Text mining, also referred to as text data mining, roughly
equivalent to text analytics,\n
refers to the process of deriving high-quality information from text.
High-quality information is \
typically derived through the devising of patterns and trends through
means such as statistical \
pattern learning. Text mining usually involves the process of
structuring the input text \
(usually parsing, along with the addition of some derived linguistic
features and the removal \
of others, and subsequent insertion into a database), deriving
patterns within the structured data, \
and finally evaluation and interpretation of the output. 'High
quality' in text mining usually \
refers to some combination of relevance, novelty, and interestingness.
Typical text mining tasks \
include text categorization, text clustering, concept/entity
extraction, production of granular \
taxonomies, sentiment analysis, document summarization, and entity
relation modeling \
(i.e., learning relations between named entities).Text analysis
involves information retrieval, \
lexical analysis to study word frequency distributions, pattern
recognition, tagging/annotation, \
information extraction, data mining techniques including link and
association analysis, \
visualization, and predictive analytics. The overarching goal is,
essentially, to turn text \
into data for analysis, via application of natural language processing
(NLP) and analytical \
methods.A typical application is to scan a set of documents written in
a natural language and \
```

```
either model the document set for predictive classification purposes  
or populate a database \  
or search index with the information extracted."
```

Let's now demonstrate the stop words removal process:

```
words = word_tokenize(text)  
# 2. Let us get the list of stopwords from nltk stopwords english  
corpus.  
stop_words = stopwords.words('english')  
  
print "Number of words = %d"%(len(words))  
# 3. Filter out the stop words.  
words = [w for w in words if w not in stop_words]  
print "Number of words,without stop words = %d"%(len(words))  
  
words = [w for w in words if w not in string.punctuation]  
print "Number of words,without stop words and punctuations =  
%d"%(len(words))
```

How it works...

In step 1, we will import the necessary libraries from nltk. We will need the list of English stop words, so we will import the stop word corpus. We will need to tokenize our input text into words. For this, we will import the `word_tokenize` function from the `nltk.tokenize` module.

For our input text, we took the introduction paragraph from Wikipedia on text mining, which can be found at http://en.wikipedia.org/wiki/Text_mining.

Finally, we will tokenize the input text into words using the `word_tokenize` function. The words is now a list of all the words tokenized from the input. Let's look at the output of the print function, where we will print the length of the words list:

```
Number of words = 259
```

We have a total of 259 words in our list.

In step 2, we will compile a list of the English stop words in a list called `stop_words`.

In step 2, we will use a list comprehension to get a final list of the words; only those words that are not in the stop word list that we created in step 2. This way, we can remove the stop words from our input. Let's now look at the output of our print statement, where we will print the final list where the stop words have been removed:

```
Number of words, without stop words = 195
```

You can see that we chopped off nearly 64 words from our input text, which were the stop words.

There's more...

Stop words are not limited to proper English words. It's contextual, depending on the application in hand and how you want to program your system. Ideally, if we are not interested in special characters, we can include them in our stop word list. Let's look at the following code:

```
import string
words = [w for w in words if w not in string.punctuation]
print "Number of words, without stop words and punctuations = %d" %(len(words))
```

Here, we will run another list comprehension in order to remove punctuations from our words. Now, the output looks as follows:

```
Number of words, without stop words and punctuations = 156
```



Remember that stop word removal is contextual and based on the application. If you are working on a sentiment analysis application on mobile or chat room text, emoticons are highly useful. You don't remove them as they form a very good feature set for the downstream machine learning application.

Typically, in a document, the frequency of stop words is very high. However, there may be other words in your corpus that may have a very high frequency. Based on your context, you can add them to your stop word list.

See also

- ▶ *Performing Tokenization recipe in Chapter 3, Analyzing Data - Explore & Wrangle*
- ▶ *List generation recipe in Chapter 1, Using Python for Data Science*

Stemming the words

In this we will see how to stem the word.

Getting ready

Standardization of the text is a different beast and we need different tools to tame it. In this section, we will look into how we can convert words to their base forms in order to bring consistency to our processing. We will start with traditional ways that include stemming and lemmatization. English grammar dictates how certain words are used in sentences. For example, perform, performing, and performs indicate the same action; they appear in different sentences based on the grammar rules.

The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form.

Introduction to Information Retrieval By Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze

Let's look into how we can perform word stemming using Python NLTK. NLTK provides us with a rich set of functions that can help us do the stemming pretty easily:

```
>>> import nltk.stem  
>>> dir(nltk.stem)  
['ISRIStemmer', 'LancasterStemmer', 'PorterStemmer', 'RSLPStemmer',  
'RegexpStemmer', 'SnowballStemmer', 'StemmerI', 'WordNetLemmatizer',  
'__builtins__', '__doc__', '__file__', '__name__', '__package__',  
'__path__', 'api', 'isri', 'lancaster', 'porter', 'regexp', 'rslp',  
'snowball', 'wordnet']  
>>>
```

We can see the list of functions in the module, and for our interest, we have the following stemmers:

- ▶ Porter – porter stemmer
- ▶ Lancaster – Lancaster stemmer
- ▶ Snowball – snowball stemmer

Porter is the most commonly used stemmer. The algorithm is not very aggressive when moving words to their root form.

Snowball is an improvement over porter. It is also faster than porter in terms of the computational time.

Lancaster is the most aggressive stemmer. With porter and snowball, the final word tokens would still be readable by humans, but with Lancaster, it is not readable. It's the fastest of the trio.

In this recipe, we will use some of them to see how the stemming of words can be performed.

How to do it...

To begin with, let's load the necessary libraries and declare the dataset against which we would want to demonstrate stemming:

```
# Load Libraries
from nltk import stem

#1. small input to figure out how the three stemmers perform.
input_words = ['movies', 'dogs', 'planes', 'flowers', 'flies', 'fries', 'fr
y', 'weeks', 'planted', 'running', 'throttle']
```

Let's jump into the different stemming algorithms, as follows:

```
#2. Porter Stemming
porter = stem.porter.PorterStemmer()
p_words = [porter.stem(w) for w in input_words]
print p_words

#3. Lancaster Stemming
lancaster = stem.lancaster.LancasterStemmer()
l_words = [lancaster.stem(w) for w in input_words]
print l_words

#4. Snowball stemming
snowball = stem.snowball.EnglishStemmer()
s_words = [snowball.stem(w) for w in input_words]
print s_words

wordnet_lemm = stem.WordNetLemmatizer()
wn_words = [wordnet_lemm.lemmatize(w) for w in input_words]
print wn_words
```

How it works...

In step 1, we will import the stem module from nltk. We will also create a list of words that we want to stem. If you observe carefully, the words have been chosen to have different suffixes, including s, ies, ed, ing, and so on. Additionally, there are some words in their root form already, such as throttle and fry. The idea is to see how the stemming algorithm treats them.

Steps 2, 3, and 4 are very similar; we will invoke the porter, lancaster, and snowball stemmers on the input and print the output. We will use a list comprehension to apply these words to our input and finally, print the output. Let's look at the print output to understand the effect of stemming:

```
[u'movi', u'dog', u'plane', u'flower', u'fli', u'fri', u'fri',
 u'week', u'plant', u'run', u'thrott1']
```

This is the output from step 2. Porter stemming was applied to our input words. We can see that the words with the suffixes ies, s, ed , and ing have been reduced to their root forms:

```
Movies - movi
Dogs - dog
Planes - plane
Running - run and so on.
```

It's interesting to note that throttle is changed to throttle.

In step 3, we will print the output of lancaster, which is as follows:

```
[u'movy', 'dog', 'plan', 'flow', 'fli', 'fri', 'fry', 'week', 'plant',
 'run', 'throttle']
```

The word throttle has been left as it is. Note what has happened to movies.

Similarly, let's look at the output produced by the snowball stemmer in step 4:

```
[u'movi', u'dog', u'plane', u'flower', u'fli', u'fri', u'fri',
 u'week', u'plant', u'run', u'thrott1']
```

The output is pretty similar to the porter stemmer.

There's more...

All the three algorithms are pretty involved; going into the details of these algorithms is beyond the scope of this book. I will recommend you to look to the web for more details on these algorithms.

For details of the porter and snowball stemmers, refer to the following link:

<http://snowball.tartarus.org/algorithms/porter/stemmer.html>

See also

- ▶ *List Comprehension* recipe in Chapter 1, Using Python for Data Science

Performing word lemmatization

In this we will learn how to perform word lemmatization.

Getting ready

Stemming is a heuristic process, which goes about chopping the word suffixes in order to get to the root form of the word. In the previous recipe, we saw that it may end up chopping even the right words, that is, chopping the derivational affixes.

See the following Wikipedia link for the derivational patterns:

http://en.wikipedia.org/wiki/Morphological_derivation#Derivational_patterns

On the other hand, lemmatization uses a morphological analysis and vocabulary to get the lemma of a word. It tries to change only the inflectional endings and give the base word from a dictionary.

See Wikipedia for more information on inflection at <http://en.wikipedia.org/wiki/Inflection>.

In this recipe, we will use NLTK's WordNetLemmatizer.

How to do it...

To begin with, we will load the necessary libraries. Once again, as we did in the previous recipes, we will prepare a text input in order to demonstrate lemmatization. We will then proceed to implement lemmatization in the following way:

```
# Load Libraries
from nltk import stem

#1. small input to figure out how the three stemmers perform.
input_words = ['movies', 'dogs', 'planes', 'flowers', 'flies', 'fries', 'fr
y', 'weeks',
               'planted', 'running', 'throttle']

#2. Perform lemmatization.
wordnet_lemm = stem.WordNetLemmatizer()
```

```
wn_words = [wordnet_lemm.lemmatize(w) for w in input_words]
print wn_words
```

How it works...

Step 1 is very similar to our stemming recipe. We will provide the input. In step 2, we will do the lemmatization. This lemmatizer uses Wordnet's built-in morphy-function.

<https://wordnet.princeton.edu/man/morphy.7WN.html>

Let's look at the output from the print statement:

```
[u'movie', u'dog', u'plane', u'flower', u'fly', u'fry', 'fry',
 u'week', 'planted', 'running', 'throttle']
```

The first thing to strike is the word movie. You can see that it has got this right. Porter and the other algorithms had chopped the last letter e.

There's more...

Let's look into a small example using lemmatizer:

```
>>> wordnet_lemm.lemmatize('running')
'running'
>>> porter.stem('running')
u'run'
>>> lancaster.stem('running')
'run'
>>> snowball.stem('running')
u'run'
```

The word running should ideally be run and our lemmatizer should have gotten it right. We can see that it has not made any changes to running. However, our heuristic-based stemmers have got it right! Then, what has gone wrong with our lemmatizer?



By default, the lemmatizer assumes that the input is a noun; this can be rectified by passing the POS tag of the word to our lemmatizer, as follows:

```
>>> wordnet_lemm.lemmatize('running', 'v') u'run'
```

See also

- ▶ *Performing Tokenization recipe in Chapter 3, Analyzing Data - Explore & Wrangle*

Representing the text as a bag of words

In this we will learn how represent the text as a bag of words.

Getting ready

In order to do machine learning on text, we will need to convert the text to numerical feature vectors. In this section, we will look into the bag of words representation, where the text is converted to numerical vectors and the column names are the underlying words and values can be either of thw following points:

- ▶ Binary, which indicates whether the word is present/absent in the given document
- ▶ Frequency, which indicates the count of the word in the given document
- ▶ TFIDF, which is a score that we will cover subsequently

Bag of words is the most frequent way of representing the text. As the name suggests, the order of words is ignored and only the presence/absence of words are key to this representation. It is a two-step process, as follows:

1. For every word in the document that is present in the training set, we will assign an integer and store this as a dictionary.
2. For every document, we will create a vector. The columns of the vectors are the actual words itself. They form the features. The values of the cell are binary, frequency, or TFIDF.

How to do it...

Let's load the necessary libraries and prepare the dataset for the demonstration of bag of words:

```
# Load Libraries
from nltk.tokenize import sent_tokenize
from sklearn.feature_extraction.text import CountVectorizer
from nltk.corpus import stopwords

# 1. Our input text, we use the same input which we had used in
stop word removal recipe.
text = "Text mining, also referred to as text data mining, roughly
equivalent to text analytics,\n
```

refers to the process of deriving high-quality information from text. High-quality information is \\ typically derived through the devising of patterns and trends through means such as statistical \\ pattern learning. Text mining usually involves the process of structuring the input text \\ (usually parsing, along with the addition of some derived linguistic features and the removal \\ of others, and subsequent insertion into a database), deriving patterns within the structured data, \\ and finally evaluation and interpretation of the output. 'High quality' in text mining usually \\ refers to some combination of relevance, novelty, and interestingness. Typical text mining tasks \\ include text categorization, text clustering, concept/entity extraction, production of granular \\ taxonomies, sentiment analysis, document summarization, and entity relation modeling \\ (i.e., learning relations between named entities). Text analysis involves information retrieval, \\ lexical analysis to study word frequency distributions, pattern recognition, tagging/annotation, \\ information extraction, data mining techniques including link and association analysis, \\ visualization, and predictive analytics. The overarching goal is, essentially, to turn text \\ into data for analysis, via application of natural language processing (NLP) and analytical \\ methods. A typical application is to scan a set of documents written in a natural language and \\ either model the document set for predictive classification purposes or populate a database \\ or search index with the information extracted."

Let's jump into how to transform the text into a bag of words representation:

```
#2.Let us divide the given text into sentences
sentences = sent_tokenize(text)

#3.Let us write the code to generate feature vectors.
count_v = CountVectorizer()
tdm = count_v.fit_transform(sentences)

# While creating a mapping from words to feature indices, we can
# ignore
# some words by providing a stop word list.
```

```
stop_words = stopwords.words('english')
count_v_sw = CountVectorizer(stop_words=stop_words)
sw_tdm = count_v_sw.fit_transform(sentences)

# Use ngrams
count_v_ngram = CountVectorizer(stop_words=stop_words,ngram_
range=(1,2))
ngram_tdm = count_v_ngram.fit_transform(sentences)
```

How it works...

In step 1, we will define the input. This is the same input that we used for the stop word removal recipe. In step 2, we will import the sentence tokenizer and tokenize the given input into sentences. We will treat every sentence here as a document:



Depending on your application, the notion of a document can change. In this case, our sentence is considered as a document. In some cases, we can also treat a paragraph as a document. In web page mining, a single web page can be treated as a document or parts of the web page separated by the <p> tags can also be treated as a document.

```
>>> len(sentences)
6
>>>
```

If we print the length of the sentence list, we will get six, and so in our case, we have six documents.

In step 3, we will import CountVectorizer from the scikitlearn.feature_extraction.text package. It converts a collection of documents—in this case, a list of sentences—to a matrix, where the rows are sentences and the columns are the words in these sentences. The count of these words are inserted in the value of these cells.

We will transform the list of sentences into a term document matrix using `CountVectorizer`. Let's dissect the output one by one. First, we will look into `count_v`, which is a `CountVectorizer` object. We had mentioned in the introduction that we need to build a dictionary of all the words in the given text. The `vocabulary_` of `count_v` attribute provides us with the list of words and their associated IDs or feature indices:

```
>>> count_v.vocabulary_
{u'nlp': 65, u'named': 63, u'concept': 16, u'interpretation': 49,
3, u'classification': 13, u'text': 107, u'into': 50, u'within': 11
27, u'structuring': 98, u'via': 116, u'through': 109, u'statistica
': 101, u'quality': 81, u'linguistic': 56, u'clustering': 14, u'vi
117, u'categorization': 12, u'from': 37, u'to': 110, u'addition':
d': 97, u'relations': 86, u'removal': 88, u'granular': 39, u'entit
nally': 34, u'production': 79, u'tagging': 103, u'relevance': 87,
u'include': 42, u'mining': 60, u'combination': 15, u'means': 58, u
u'link': 57, u'devising': 21, u'parsing': 72, u'with': 118, u'asso
u'extracted': 31, u'document': 23, u'word': 120, u'overarching': 7
u'as': 9, u'either': 25, u'analytical': 4, u'including': 43, u'ref
and': 6, u'predictive': 76, u'set': 94, u'methods': 59, u'scan': 9
77, u'the': 108, u'is': 52, u'some': 95, u'purposes': 80, u'roughl
```

This dictionary can be retrieved using the `vocabulary_` attribute. This is a map of the terms in order to feature indices. We can also use the following function to get the list of words (features):

```
>>> count_v.get_feature_names()
```

Let's now move on to look at `tdm`, which is the object that we received after transforming the given input using `CountVectorizer`:

```
>>> type(tdm)
<class 'scipy.sparse.csr.csr_matrix'>
>>>
```

As you can see, `tdm` is a sparse matrix object. Refer to the following link to understand more about the sparse matrix representation:

http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.sparse.csr_matrix.html

We can look into the shape of this object and also inspect some of the elements, as follows:

```
>>> tdm.get_shape()
(6, 122)
>>> tdm.indices
array([2, 2, 0, 4, 5, 5, 0, 4, 1, 2, 3, 4, 5, 4, 5, 0, 1, 4, 4, 4, 4, 5, 4, 3,
       4, 0, 2, 4, 5, 2, 5, 1, 2, 0, 2, 1, 4, 4, 5, 5, 5, 4, 4, 0, 5, 2, 5,
       4, 2, 2, 5, 4, 0, 5, 4, 0, 1, 3, 3, 5, 4, 4, 5, 0, 1, 4, 5, 2, 2, 3,
       2, 2, 5, 2, 4, 1, 5, 5, 1, 4, 4, 2, 4, 1, 5, 0, 2, 3, 4, 5, 4, 4, 5,
       5, 3, 0, 1, 2, 3, 4, 5, 5, 2, 2, 5, 2, 1, 4, 1, 2, 5, 4, 5, 0, 2, 5,
       4, 5, 0, 1, 3, 4, 0, 0, 3, 4, 4, 3, 2, 4, 0, 5, 5, 4, 5, 2, 3, 1, 2,
       2, 4, 2, 1, 4, 4, 4, 4, 0, 2, 3, 4, 5, 0, 1, 2, 5, 1, 0, 3, 4, 5,
       1, 5, 4, 5, 1, 2, 3, 5, 4, 2, 5, 2, 4, 5])
>>> tdm.data
array([1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 4, 1, 3, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2, 1, 1, 1,
       1, 1, 1, 1, 4, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 2, 1, 1, 4, 1, 1, 1, 6, 3, 2, 3, 1, 1, 2,
       1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> tdm.indptr
array([ 0,    1,    2,    3,    5,    6,    8,   13,   14,   15,   17,   18,   19,
       20,   21,   22,   23,   24,   28,   30,   32,   34,   35,   36,   38,   39,
       40,   41,   42,   43,   44,   45,   46,   47,   48,   49,   50,   51,   52,
       53,   54,   57,   59,   60,   61,   62,   66,   67,   68,   69,   70,   72,
       74,   76,   77,   79,   80,   81,   82,   83,   84,   88,   89,   90,   91,
       92,   93,   94,  100,  101,  102,  103,  104,  105,  107,  109,  110,  112,
      114,  115,  116,  117,  120,  121,  122,  124,  125,  126,  127,  128,  129,
      130,  131,  132,  133,  134,  136,  137,  138,  139,  140,  141,  142,  143,
      144,  145,  146,  147,  152,  156,  157,  161,  162,  163,  165,  166,  168,
      169,  170,  172,  173,  174,  175])
```

We can see that the shape of the matrix is 6×122 . We have six documents, that is, sentences in our context and 122 words that form the vocabulary. Note that this is a sparse matrix representation; as all the sentences will not have all the words, a lot of the cell values will have zero as an entry and hence, we will print only the indices that have non-zero entries.

From `tdm.indptr`, we know that document 1's entry starts from 0 and ends at 18 in the `tdm.data` and `tdm.indices` arrays, as follows:

```
>>> tdm.data[0:17]
array([4, 2, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],  
      dtype=int64)
>>> tdm.indices[0:17]
```

```
>>> array([107, 60, 2, 83, 110, 9, 17, 90, 28, 5, 84, 108,
77,
       67, 20, 40, 81])
>>>
```

We can verify this in the following way:

```
>>> count_v.get_feature_names()[107]
u'text'
>>> count_v.get_feature_names()[60]
u'mining'
```

We can see that 107, which corresponds to the word text, has occurred four times in the first sentence, and similarly, mining has occurred once. Thus, in this recipe, we converted a given text into a feature vector, where the features are words.

There's more...

The CountVectorizer class has a lot of other features to offer in order to transform the text into feature vectors. Let's look at some of them:

```
>>> count_v.get_params()
{'binary': False, 'lowercase': True, 'stop_words': None, 'vocabulary': None,
'tokenizer': None, 'decode_error': u'strict', 'dtype': <type
'numpy.int64'>, 'charset_error': None, 'charset': None, 'analyzer': u'word',
'encoding': u'utf-8', 'ngram_range': (1, 1), 'max_df': 1.0,
'min_df': 1, 'max_features': None, 'input': u'content', 'strip_accents': None,
'token_pattern': u'(?u)\\b\\w+\\b', 'preprocessor': None}
>>>
```

The first one is binary, which is set to `False`; we can also have it set to `True`. Then, the final matrix would not have the count but will have one or zero, based on the presence or absence of the word in the document.

The lowercase is set to `True` by default; the input text is transformed into lowercase before the mapping of the words to feature indices is performed.

While creating a mapping of the words to feature indices, we can ignore some words by providing a stop word list. Observe the following example:

```
from nltk.corpus import stopwords
stop_words = stopwords.words('english')

count_v = CountVectorizer(stop_words=stop_words)
sw_tdm = count_v.fit_transform(sentences)
```

If we print the size of the vocabulary that has been built, we can see the following:

```
>>> len(count_v_sw.vocabulary_)  
106  
>>>
```

We can see that we have 106 now as compared to 122 that we had before.

We can also give a fixed set of vocabulary to CountVectorizer. The final sparse matrix columns will be only from these fixed sets and anything that is not in this set will be ignored.

The next interesting parameter is the ngram range. You can see that a tuple (1,1) has been passed. This ensures that only one grams or single words are used while creating a feature set. For example, this can be changed to (1,2), which tells CountVectorizer to create both unigrams and bigrams. Let's look at the following code and the output:

```
count_v_ngram = CountVectorizer(stop_words=stop_words, ngram_  
range=(1,2))  
ngram_tdm = count_v_ngram.fit_transform(sentences)
```

Both the unigrams and bigrams are now a part of our feature set.

I will leave you to explore the other parameters. The documentation for these parameters is available at the following link:

http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

See also

- ▶ *Using Dictionaries recipe in Chapter 1, Using Python for Data Science*
- ▶ *Removing Stop words, Stemming of words, Lemmatization of words recipe in Chapter 3, Analyzing Data - Explore & Wrangle*

Calculating term frequencies and inverse document frequencies

In this we will learn how to calculate term frequencies and inverse document frequencies.

Getting ready

Occurrences and counts are good as feature values, but they suffer from some problems. Let's say that we have four documents of unequal length. This will give a higher weightage to the terms in the longer documents than those in the shorter ones. So, instead of using the plain vanilla occurrence, we will normalize it; we will divide the number of occurrences of a word in a document by the total number of words in the document. This metric is called term frequencies. Term frequency is also not without problems. There are words that will occur in many documents. These words would dominate the feature vector but they are not informative enough to distinguish the documents in the corpus. Before we look into a new metric that can avoid this problem, let's define document frequency. Similar to word frequency, which is local with respect to a document, we can calculate a score called document frequency, which is the number of documents that the word occurs in the corpus divided by the total number of documents in the corpus.

The final metric that we will use for the words is the product of the term frequency and the inverse of the document frequency. This is called the TFIDF score.

How to do it...

Load the necessary libraries and declare the input data that will be used for the demonstration of term frequencies and inverse document frequencies:

```
# Load Libraries
from nltk.tokenize import sent_tokenize
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import CountVectorizer

# 1.      We create an input document as in the previous recipe.

text = "Text mining, also referred to as text data mining, roughly
equivalent to text analytics,\nrefers to the process of deriving high-quality information from text.
High-quality information is \
typically derived through the devising of patterns and trends through
means such as statistical \
pattern learning. Text mining usually involves the process of
structuring the input text \
(usually parsing, along with the addition of some derived linguistic
features and the removal \
of others, and subsequent insertion into a database), deriving
patterns within the structured data, \
```

and finally evaluation and interpretation of the output. 'High quality' in text mining usually \\ refers to some combination of relevance, novelty, and interestingness. Typical text mining tasks \\ include text categorization, text clustering, concept/entity extraction, production of granular \\ taxonomies, sentiment analysis, document summarization, and entity relation modeling \\ (i.e., learning relations between named entities). Text analysis involves information retrieval, \\ lexical analysis to study word frequency distributions, pattern recognition, tagging/annotation, \\ information extraction, data mining techniques including link and association analysis, \\ visualization, and predictive analytics. The overarching goal is, essentially, to turn text \\ into data for analysis, via application of natural language processing (NLP) and analytical \\ methods. A typical application is to scan a set of documents written in a natural language and \\ either model the document set for predictive classification purposes or populate a database \\ or search index with the information extracted."

Let's see how to find the term frequency and inverse document frequency:

```
# 2.      Let us extract the sentences.  
sentences = sent_tokenize(text)  
  
# 3.      Create a matrix of term document frequency.  
stop_words = stopwords.words('english')  
  
count_v = CountVectorizer(stop_words=stop_words)  
tdm = count_v.fit_transform(sentences)  
  
#4. Calculate the TFIDF score.  
tfidf = TfidfTransformer()  
tdm_tfidf = tfidf.fit_transform(tdm)
```

How it works...

Steps 1, 2, and 3 are the same as the previous recipe. Let's look at step 4, where we will pass the output of step 3 in order to calculate the TFIDF score:

```
>>> type(tdm)  
<class 'scipy.sparse.csr.csr_matrix'>  
>>>
```

Tdm is a sparse matrix. Now, let's look at the values of these matrices, using indices, data, and index pointer:

```
>>> tdm_tfidf.data
array([ 0.54849062,  0.26764689,  0.21947428,  0.26764689,  0.18529527,
       0.21947428,  0.31756793,  0.15878397,  0.18529527,  0.26764689,
       0.21947428,  0.15878397,  0.21947428,  0.26764689,  0.33332883,
       0.33332883,  0.33332883,  0.23076769,  0.27333441,  0.27333441,
       0.33332883,  0.27333441,  0.19775038,  0.23076769,  0.33332883,
       0.27333441,  0.19811475,  0.32491385,  0.20299896,  0.19811475,
       0.19811475,  0.19811475,  0.19811475,  0.16245693,  0.16245693,
       0.19811475,  0.19811475,  0.19811475,  0.11753339,  0.19811475,
       0.16245693,  0.19811475,  0.19811475,  0.19811475,  0.19811475,
       0.19811475,  0.19811475,  0.16245693,  0.16245693,  0.16245693,
       0.11753339,  0.19811475,  0.19811475,  0.31177054,  0.19478731,
       0.38020134,  0.31177054,  0.26321811,  0.38020134,  0.22555792,
       0.38020134,  0.26321811,  0.38020134,  0.12927948,  0.12927948,
       0.10601102,  0.26493333,  0.12927948,  0.12927948,  0.12927948,
       0.12927948,  0.12927948,  0.12927948,  0.12927948,  0.12927948,
       0.12927948,  0.12927948,  0.12927948,  0.12927948,  0.10601102,
       0.10601102,  0.12927948,  0.12927948,  0.15339247,  0.12927948,
       0.12927948,  0.10601102,  0.10601102,  0.15339247,  0.12927948,
       0.12927948,  0.12927948,  0.12927948,  0.25855896,  0.25855896,
       0.12927948,  0.10601102,  0.12927948,  0.07669623,  0.12927948,
       0.12927948,  0.12927948,  0.12927948,  0.12927948,  0.10601102,
       0.42404408,  0.15746858,  0.15746858,  0.12912649,  0.15746858,
       0.08067536,  0.31493717,  0.15746858,  0.15746858,  0.15746858,
       0.15746858,  0.12912649,  0.15746858,  0.15746858,  0.15746858,
       0.31493717,  0.15746858,  0.15746858,  0.31493717,  0.09341968,
       0.15746858,  0.15746858,  0.15746858,  0.15746858,  0.15746858,
       0.15746858,  0.12912649,  0.09341968,  0.15746858,
       0.31493717,  0.15746858,  0.12912649])
>>> tdm_tfidf.indices
array([ 95,  80,  74,  73,  71,  67,  52,  39,  35,  25,  17,  14,  5,
       2,  99,  96,  85,  71,  64,  63,  50,  46,  39,  35,  18,  16,
      103,  100,  95,  89,  87,  86,  78,  67,  64,  62,  60,  59,  52,
      48,  44,  43,  41,  40,  31,  30,  27,  17,  16,  15,  14,  1,
       0,  100,  95,  77,  74,  71,  58,  52,  42,  35,  12,  104,  102,
      98,  95,  94,  93,  92,  91,  90,  88,  83,  79,  76,  75,  72,
      69,  66,  63,  55,  54,  52,  49,  47,  46,  44,  39,  37,  36,
      34,  32,  29,  24,  23,  20,  19,  14,  13,  11,  9,  8,  6,
       5,   3,  105, 101,  98,  97,  95,  84,  82,  81,  70,  68,  66,
      65,  61,  57,  56,  53,  51,  45,  39,  38,  33,  28,  26,  22,
      21,  20,  15,  14,  10,   7,   4,    3])
>>> tdm_tfidf.indptr
```

The data shows the values, we don't have the occurrences, but the normalized TFIDF score for the words.

There's more...

Once again, we can delve deeper into the TFIDF transformer by looking into the parameters that can be passed:

```
>>> tfidf.get_params()  
{'use_idf': True, 'smooth_idf': True, 'sublinear_tf': False, 'norm':  
u'l2'}  
>>>
```

The documentation for this is available at http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html.

4

Data Analysis – Deep Dive

In this chapter, we will cover the following topics:

- ▶ Extracting the principal components
- ▶ Using Kernel PCA
- ▶ Extracting features using Singular Value Decomposition
- ▶ Reducing the data dimension with Random Projection
- ▶ Decomposing Feature matrices using **NMF (Non-negative Matrix Factorization)**

Introduction

In this chapter, we will look at recipes dealing with dimensionality reduction. In the previous chapter, we looked at how to surf through the data to understand its characteristics in order to put it to meaningful use. We restricted our discussions to only bivariate data. Imagine a dataset with hundreds of columns; how do we proceed with the analysis of the data characteristics of such a large dimensional dataset? We need efficient tools to handle this hurdle as we work our way through the data.

Today, high-dimensional data is everywhere. Consider building a product recommendation engine for a moderately-sized e-commerce website. Even with the range of thousands of products, the number of variables to consider very high. Bioinformatics is another area with very high-dimensional data. Gene expression are microarray datasets could contain tens of thousands of dimensions.

If your task at hand is to either explore the data or prepare the data for an algorithm, the high dimensionality, popularly called the *curse of dimensionality*, is a big roadblock. We need efficient methods to handle this. Additionally, the complexity of many existing data mining algorithms increases exponentially with the increase in the number of dimensions. With increasing dimensions, the algorithms become computationally infeasible and thus inapplicable in many applications.

Dimensionality reduction techniques preserve the structure of the data as much as possible while reducing the number of dimensions. Thus, in the reduced feature space, the execution time of the algorithms is reduced as we have lower dimensions. As the structure of data is preserved, the results obtained can be a reliable approximation of the original data space. By preserving the structure, we mean two things; the first is not tampering with the variations in the original dataset and the second is preserving the distance between the data vectors in the new projected space.

Matrix Decomposition:

Matrix decomposition yields several techniques for dimensionality reduction. Our data is typically a matrix with the instances in rows and features in columns. In the previous recipes, we have been storing our data as NumPy matrices all the way. For example, in the Iris dataset, our tuples or data instances were represented as rows and the features, which included sepal and petal width and length, were the columns of the matrix.

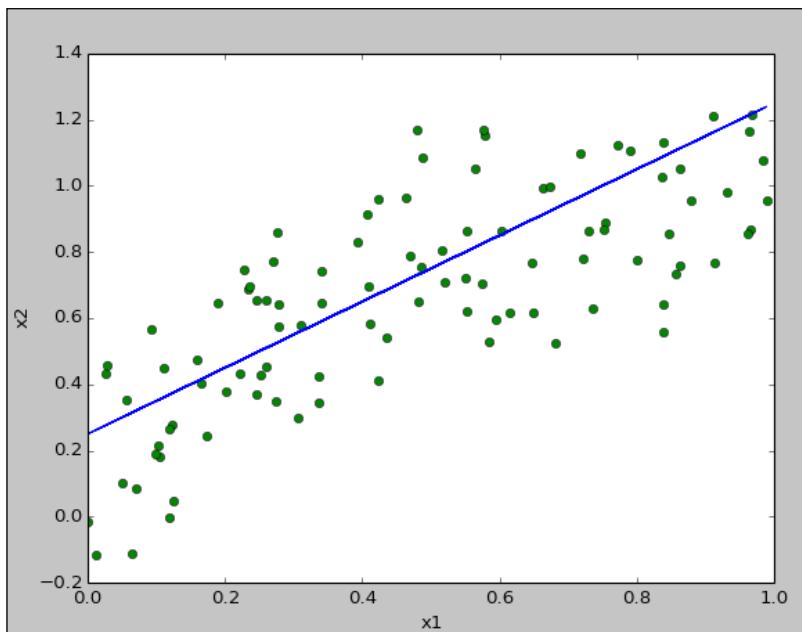
Matrix decomposition is a way of expressing a matrix. Say that A is a product of two other matrices, B and C. The matrix B is supposed to contain vectors that can explain the direction of variation in the data. The matrix C is supposed to contain the magnitude of this variation. Thus, our original matrix A is now expressed as a linear combination of B and C.

The techniques that we will see in the coming sections exploit matrix decomposition in order to tackle the dimensionality reduction. There are methods that insist that the basic vectors have to be orthogonal to each other, such as the principal component analysis, and there are some that don't insist on this requirement, such as dictionary learning.

Let's buckle up and see some of these techniques in action in this chapter.

Extracting the principal components

The first technique that we will look at is the **Principal Component Analysis (PCA)**. PCA is an unsupervised method. In multivariate problems, PCA is used to reduce the dimension of the data with minimal information loss, in other words, retaining the maximum variation in the data. By variation, we mean the direction in which the data is dispersed to the maximum. Let's look at the following plot:



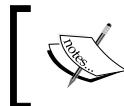
We have a scatter plot of two variables, x_1 and x_2 . The diagonal line indicates the maximum variation. By using PCA, our intent is to capture this direction of the variation. So, instead of using the direction of two variables, x_1 and x_2 , to represent this data, the quest is to find a vector represented by the blue line and represent the data with only this vector. Essentially we want to reduce the dimension of the data from two to one.

We will leverage the mathematical tools Eigenvalues and Eigenvectors to find this blue line vector.

We saw in the previous chapter that the variance measures the amount of dispersion or spread in the data. What we saw was an example in one dimension. In case of more than one dimension it is easy to express correlation among the variables as a matrix, called as Covariance matrix. When the values of the Covariance matrix are normalized by standard deviation we get a Correlation matrix. In our case, the covariance matrix is a 2×2 matrix for two variables, x_1 and x_2 , and it measures how much these two variables move in the same direction or generally vary together.

When we perform Eigenvalue decomposition, that is, get the Eigenvectors and Eigenvalues of the covariance matrix, the principal Eigenvector, which is the vector with the largest Eigenvalue, is in the direction of the maximum variance in the original data.

In our example, this should be the vector that is represented by the blue line in our graph. We will then proceed to project our input data in this blue line vector in order to get the reduced dimension.



With a dataset ($n \times m$) with n instances and m dimensions, PCA projects it onto a smaller subspace ($n \times d$), where $d \ll m$.
A point to note is that PCA is computationally very expensive.



PCA can be performed on both the covariance and correlation matrix. Remember when a Covariance matrix of a dataset with unevenly scaled datasets is used in PCA, the results may not be very useful. Curious readers can refer to the Book A First Course in Multivariate Statistics by Bernard Flury on the topic of using either correlation or covariance matrix for PCA.
<http://www.springer.com/us/book/9780387982069>.

Getting ready

Let's use the Iris dataset to understand how to use PCA efficiently in reducing the dimension of the dataset. The Iris dataset contains measurements for 150 iris flowers from three different species.

The three classes in the Iris dataset are as follows:

- ▶ Iris Setosa
- ▶ Iris Versicolor
- ▶ Iris Virginica

The following are the four features in the Iris dataset:

- ▶ The sepal length in cm
- ▶ The sepal width in cm
- ▶ The petal length in cm
- ▶ The petal width in cm

Can we use, say, two columns instead of all the four to express most of the variations in the data? Our quest is to reduce the dimension of the data. In this case, our instances have four columns. Let's say that we are building a classifier to predict the type of flower with a new instance; can we do this task using instances in the reduced dimension space? Can we reduce the number of columns from four to two and still achieve a good accuracy for our classifier?

PCA is done using the following steps:

1. Standardize the dataset to have a zero mean value.
2. Find the correlation matrix for the dataset and unit standard deviation value.
3. Reduce the Correlation matrix matrix into its Eigenvectors and values.
4. Select the top nEigenvectors based on the Eigenvalues sorted in descending order.
5. Project the input Eigenvectors matrix into the new subspace.

How to do it...

Let's load the necessary libraries and call the utility function `load_iris` from scikit-learn to get the Iris dataset:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import scale
import scipy
import matplotlib.pyplot as plt

# Load Iris data
data = load_iris()
x = data['data']
y = data['target']

# Since PCA is an unsupervised method, we will not be using the target
variable y
# scale the data such that mean = 0 and standard deviation = 1
x_s = scale(x,with_mean=True,with_std=True,axis=0)

# Calculate correlation matrix
x_c = np.corrcoef(x_s.T)

# Find eigen value and eigen vector from correlation matrix
eig_val,r_eig_vec = scipy.linalg.eig(x_c)
print 'Eigen values \n%s'%(eig_val)
print '\n Eigen vectors \n%s'%(r_eig_vec)

# Select the first two eigen vectors.
w = r_eig_vec[:,0:2]

# # Project the dataset in to the dimension
# from 4 dimension to 2 using the right eignen vector
```

```
x_rd = x_s.dot(w)

# Scatter plot the new two dimensions
plt.figure(1)
plt.scatter(x_rd[:,0],x_rd[:,1],c=y)
plt.xlabel("Component 1")
plt.ylabel("Component 2")
```

Now, we will proceed to Standardize this data, with a zero mean and standard deviation of one, we will leverage the `numpyscorr_coef` function to find the correlation matrix:

```
x_s = scale(x,with_mean=True,with_std=True, axis=0)
x_c = np.corrcoef(x_s.T)
```

We will then do the Eigenvalue decomposition and project our Iris data on the first two principal Eigenvectors. Finally, we will plot the dataset in the reduced space:

```
eig_val,r_eig_vec = scipy.linalg.eig(x_c)
print 'Eigen values \n%s'%(eig_val)
print '\n Eigen vectors \n%s'%(r_eig_vec)
# Select the first two eigen vectors.
w = r_eig_vec[:,0:2]

# # Project the dataset in to the dimension
# from 4 dimension to 2 using the right eigen vector
x_rd = x_s.dot(w)

# Scatter plot the new two dimensions
plt.figure(1)
plt.scatter(x_rd[:,0],x_rd[:,1],c=y)
plt.xlabel("Component 1")
plt.ylabel("Component 2")
```

Using function `scale`. The `scale` function can perform centering, scaling and standardization. Centering is subtracting the mean value from individual values, Scaling is dividing each value by the variable's standard deviation and finally Standardization is performing centering followed by scaling. Using variables `with_mean` and `with_std` function `scale` can be used to perform all three normalization techniques.

How it works...

The Iris dataset has four columns. Though there are not many columns, it will serve our purpose. We intend to reduce the dimensionality of the Iris dataset to two from four and still retain all the information about the data.

We will load the Iris data to the `x` and `y` variables using the convenient `load_iris` function from scikit-learn. The `x` variable is our data matrix and we can inspect its shape as follows:

```
>>>x.shape  
(150, 4)  
>>>
```

We will scale the data matrix `x` to have zero mean and unit standard deviation. The rule of thumb is that if all your columns are measured in the same scale in your data and have the same unit of measurement, you don't have to scale the data. This will allow PCA to capture these basic units with the maximum variation:

```
x_s = scale(x, with_mean=True, with_std=True, axis=0)
```

We will proceed to build the correlation matrix of our input data:

The correlation matrix of n random variables X₁, ..., X_n is then × n matrix whose i, j entry is corr(X_i, X_j), Wikipedia.

We will then use the SciPy library to calculate the Eigenvalues and Eigenvectors of the matrix. Let's look at our Eigenvalues and Eigenvectors:

```
print Eigen values \n%s%(eig_val)  
print \n Eigen vectors \n%s%(r_eig_vec)
```

The output looks as follows:

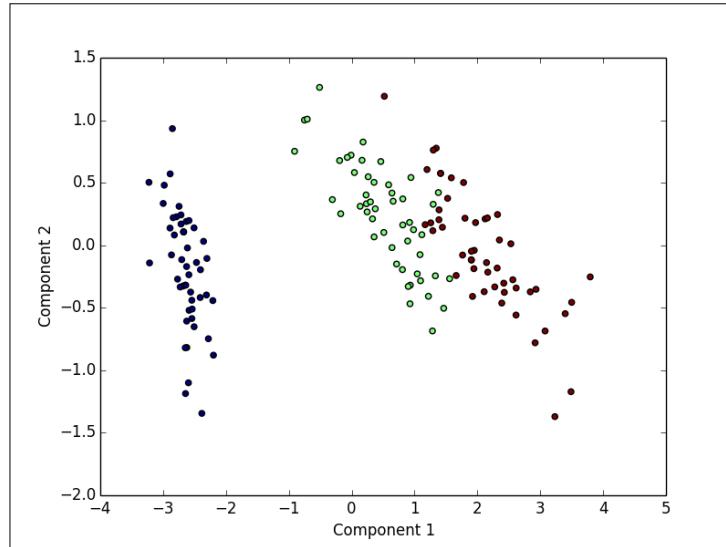
```
Eigen values  
[ 2.91081808+0.j  0.92122093+0.j  0.14735328+0.j  0.02060771+0.j]  
  
Eigen vectors  
[[ 0.52237162 -0.37231836 -0.72101681  0.26199559]  
 [-0.26335492 -0.92555649  0.24203288 -0.12413481]  
 [ 0.58125401 -0.02109478  0.14089226 -0.80115427]  
 [ 0.56561105 -0.06541577  0.6338014   0.52354627]]
```

In our case, the Eigenvalues are printed in a descending order. A key question is how many components should we choose? In the next section, we will explain a few ways of choosing the number of components.

You can see that we selected only the first two columns of our right-hand side Eigenvectors. The discrimination capability of the retained components on the `y` variable is a good test of how much information or variation is retained in the data.

We will project the data to the new reduced dimension.

Finally, we will plot the components in the x and y axes and color them by the target variable:



You can see that components 1 and 2 are able to discriminate the three classes of the iris flowers. Thus we have effectively used PCA in reducing the dimension to two from four and still able to discriminate the instances belonging to different classes of Iris flower.

There's more...

In the previous section, we said that we would outline a couple of ways to help us select how many components should we include. In our recipe, we included only two. The following are a list of ways to select the components more empirically:

1. The Eigenvalue criterion:

An Eigenvalue of one would mean that the component would explain about one variable's worth of variability. So, according to this criterion, a component should at least explain one variable's worth of variability. We can say that we will include only those Eigenvalues whose value is greater than or equal to one. Based on your data set you can set the threshold. In a very large dimensional dataset including components capable of explaining only one variable may not be very useful.

2. The proportion of the variance explained criterion:

Let's run the following code:

```
print "Component, Eigen Value, % of Variance, Cummulative %"
cum_per = 0
per_var = 0
for i,e_val in enumerate(eig_val):
```

```

per_var = round((e_val / len(eig_val)),3)
cum_per+=per_var
print ('%d, %0.2f, %0.2f, %0.2f')%(i+1, e_val, per_var*100,cum_
per*100)

```

3. The output is as follows:

Component, Eigen Value, % of Variance, Cummulative %
1, 2.91, 72.80, 72.80
2, 0.92, 23.00, 95.80
3, 0.15, 3.70, 99.50
4, 0.02, 0.50, 100.00

For each component, we printed the Eigenvalue, percentage of the variance explained by that component, and cumulative percentage value of the variance explained. For example, component 1 has an Eigenvalue of 2.91; 2.91/4 gives the percentage of the variance explained, which is 72.80%. Now, if we include the first two components, then we can explain 95.80% of the variance in the data.

The decomposition of a correlation matrix into its Eigenvectors and values is a general technique that can be applied to any matrix. In this case, we will apply it to a correlation matrix in order to understand the principal axes of data distribution, that is, axes through which the maximum variation in the data is observed.

PCA can be used either as an exploratory technique or as a data preparation technique for a downstream algorithm. Document classification dataset problems typically have very large dimensional feature vectors. PCA can be used to reduce the dimension of the dataset in order to include only the most relevant features before feeding the data to a classification algorithm.

A drawback of PCA worth mentioning here is that it is computationally expensive operation. Finally a point about numpy's corrcoeff function. The corrcoeff function will standardize your data internally as a part of its calculation. But since we want to explicitly state the reason for scaling, we have included it in our recipe.



When would PCA work?

The input dataset should have correlated columns for PCA to work effectively. Without a correlation of the input variables, PCA cannot help us.

See also

- ▶ *Performing Singular Value Decomposition* recipe in *Chapter 4, Analyzing Data - Deep Dive*

Using Kernel PCA

PCA makes an assumption that all the principal directions of variation in the data are straight lines. This is not true in a lot of real-world datasets.



PCA is limited to only those variables where the variation in the data falls in a straight line. In other words, it works only with linearly separable data.

In this section, we will look at kernel PCA, which will help us reduce the dimension of datasets where the variations in them are not straight lines. We will explicitly create such a dataset and apply kernel PCA on it.

In kernel PCA, a kernel function is applied to all the data points. This transforms the input data into kernel space. A normal PCA is performed in the kernel space.

Getting ready

We will not use the Iris dataset here, but will generate a dataset where variations are not straight lines. This way, we cannot apply a simple PCA on this dataset. Let's proceed to look at our recipe.

How to do it...

Let's load the necessary libraries. We will proceed to make a dataset using the `make_circles` function from the scikit-learn library. We will plot this data and do a normal PCA on this dataset:

```
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA
from sklearn.decomposition import KernelPCA

# Generate a dataset where the variations cannot be captured by a
# straight line.
np.random.seed(0)
x,y = make_circles(n_samples=400, factor=.2, noise=0.02)

# Plot the generated dataset
plt.close('all')
plt.figure(1)
plt.title("Original Space")
```

```
plt.scatter(x[:, 0], x[:, 1], c=y)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

# Try to fit the data using normal PCA
pca = PCA(n_components=2)
pca.fit(x)
x_pca = pca.transform(x)
```

We will then plot the first two principal components of this dataset. We will plot the dataset using only the first principal component:

```
plt.figure(2)
plt.title("PCA")
plt.scatter(x_pca[:, 0], x_pca[:, 1], c=y)
plt.xlabel("$Component_1$")
plt.ylabel("$Component_2$")

# Plot using the first component from normal pca
class_1_indx = np.where(y==0)[0]
class_2_indx = np.where(y==1)[0]

plt.figure(3)
plt.title("PCA- One component")
plt.scatter(x_pca[class_1_indx, 0], np.zeros(len(class_1_indx)), color='red')
plt.scatter(x_pca[class_2_indx, 0], np.zeros(len(class_2_indx)), color='blue')
```

Let's finish it up by performing a kernal PCA and plotting the components:

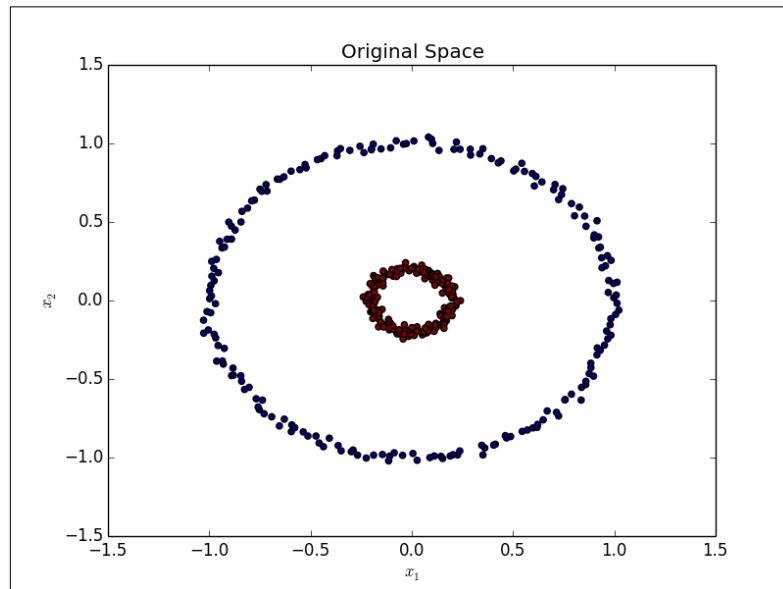
```
# Create KernelPCA object in Scikit learn, specifying a type of
# kernel as a parameter.
kPCA = KernelPCA(kernel="rbf", gamma=10)
# Perform KernelPCA
kPCA.fit(x)
x_kPCA = kPCA.transform(x)

# Plot the first two components.
plt.figure(4)
plt.title("Kernel PCA")
plt.scatter(x_kPCA[:, 0], x_kPCA[:, 1], c=y)
plt.xlabel("$Component_1$")
plt.ylabel("$Component_2$")
plt.show()
```

How it works...

In step 1, we generated a dataset using the scikit's data generation function. In this case, we used the `make_circles` function. We can create two concentric circles, a large one containing the smaller one, using this function. Each concentric circle belongs to a certain class. Thus, we created a two class problem with two concentric circles.

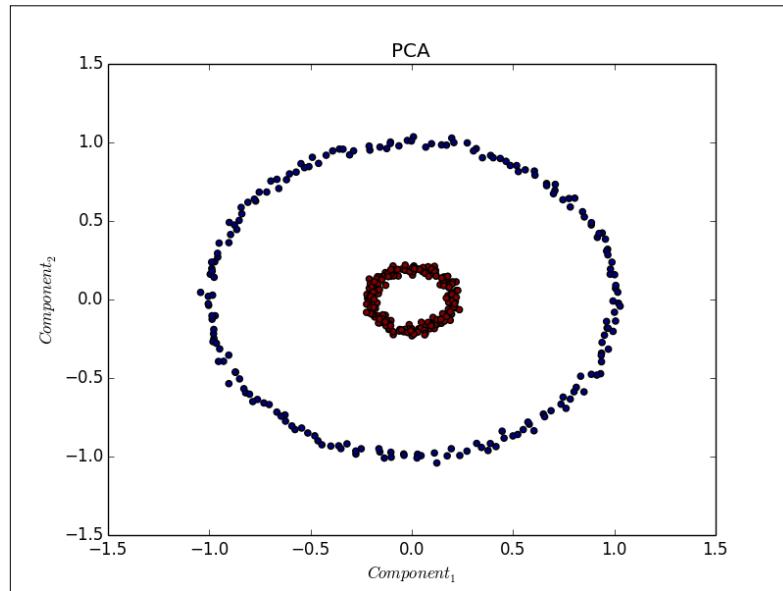
First, let's look at the data that we generated. The `make_circles` function generated a dataset of size 400 with two dimensions. A plot of the original data is as follows:



This chart describes how our data has been distributed. The outer circle belongs to class one and the inner circle belongs to class two. Is there a way we can take this data and use it with a linear classifier? We will not be able to do it. The variations in the data are not straight lines. We cannot use the normal PCA. Hence, we will resort to a kernel PCA in order to transform the data.

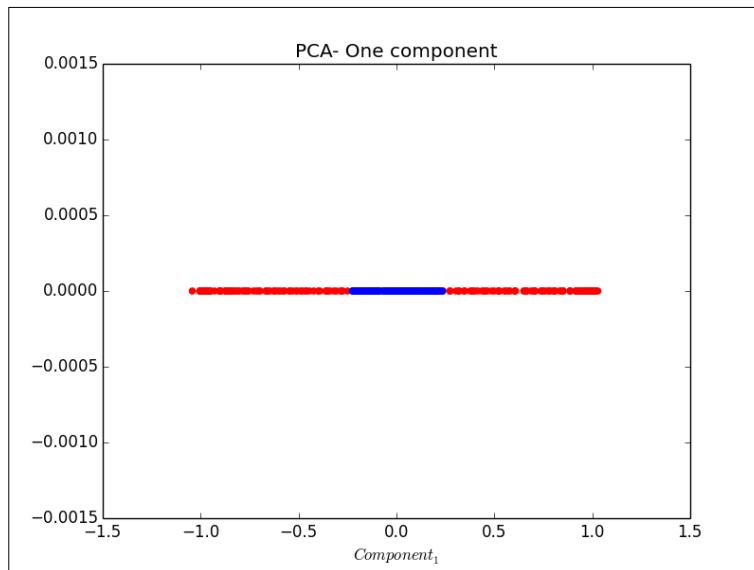
Before we venture into kernel PCA, let's see what happens if we apply a normal PCA on this dataset.

Let's look at the output plot of the first two components:



As you can see, the components of PCA are unable to distinguish between the two classes in a linear fashion.

Let's plot the first component and see its class distinguishing ability. The following graph, where we have plotted only the first component, explains how PCA is unable to differentiate the data:



The normal PCA approach is a linear projection technique that works well if the data is linearly separable. In cases where the data is not linearly separable, a nonlinear technique is required for the dimensionality reduction of the dataset.



Kernel PCA is a nonlinear technique for data reduction.



Let's proceed to create a kernel PCA object using the scikit-learn library. Here is our object creation code:

```
KernelPCA(kernel=rbf, gamma=10)
```

We selected the **Radial Basis Function (RBF)** kernel with a gamma value of ten. Gamma is the parameter of the kernel (to handle nonlinearity)—the kernel coefficient.

Before we go further, let's look at a little bit of theory about what kernels really are. As a simple definition, a kernel is a function that computes the dot product, that is, the similarity between two vectors, which are passed to it as input.

The RBFGaussian kernel is defined as follows for two points, x and x' in some input space:

$$K(x, x') = \exp\left(\gamma \cdot \|x - x'\|^2\right)$$

Where,

$$\gamma = -\frac{1}{2\sigma^2}$$

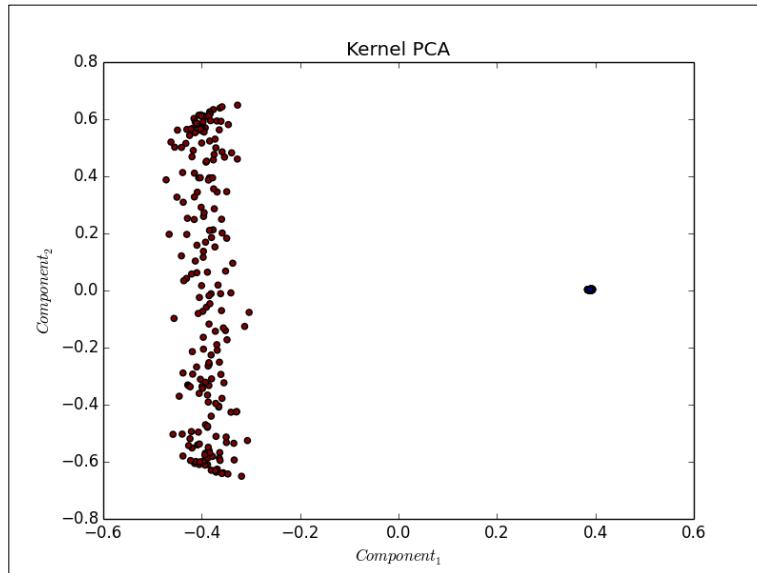
The RBF decreases with distance and takes values between 0 and 1. Hence it can be interpreted as a similarity measure. The feature space of the RBF kernel has infinite dimensions –Wikipedia.

This can be found at:

http://en.wikipedia.org/wiki/Radial_basis_function_kernel.

Let's now transform the input from the feature space into the kernel space. We will perform a PCA in the kernel space.

Finally, we will plot the first two principal components as a scatter plot. The points are colored based on their class value:



You can see in this graph that the points are linearly separated in the kernel space.

There's more...

Scikit-learn's kernel PCA object also allows other types of kernels, as follows:

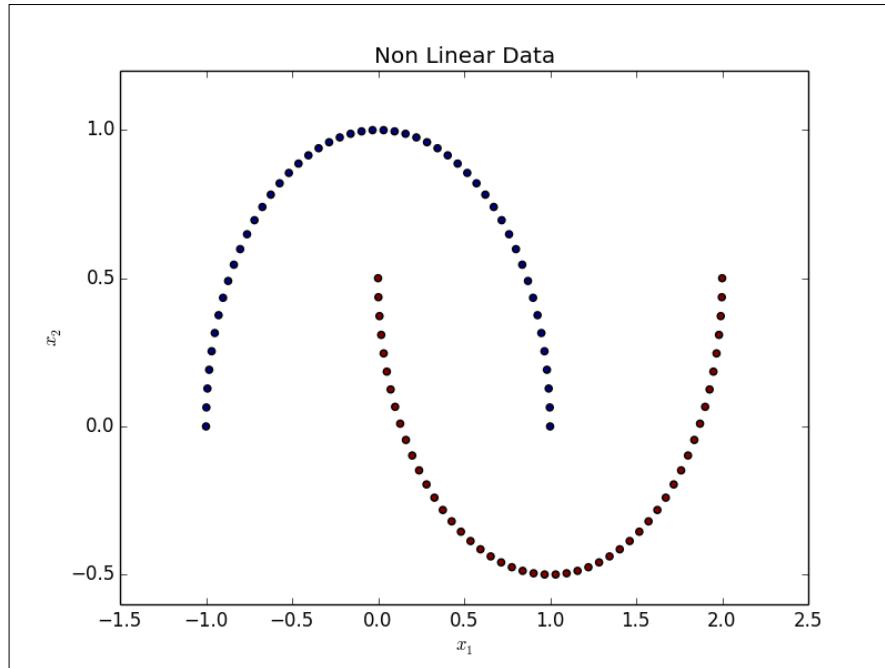
- ▶ Linear
- ▶ Polynomial
- ▶ Sigmoid
- ▶ Cosine
- ▶ Precomputed

Scikit-learn also provides other types of nonlinear data that is generated. The following is another example:

```
from sklearn.datasets import make_moons
x,y = make_moons(100)
plt.figure(5)
plt.title("Non Linear Data")
plt.scatter(x[:,0],x[:,1],c=y)
plt.xlabel("$x_1$")
```

```
plt.ylabel("$x_2$")
plt.savefig('fig-7.png')
plt.show()
```

The data plot looks as follows:



Extracting features using singular value decomposition

After our discussion on PCA and kernel PCA, we can explain dimensionality reduction in the following way:

- ▶ You can transform the correlated variables into a set of non-correlated variables. This way, we will have a less dimension explaining the relationship in the underlying data without any loss of information.
- ▶ You can find out the principal axes, which has the most data variation recorded.

Singular Value Decomposition (SVD) is yet another matrix decomposition technique that can be used to tackle the curse of the dimensionality problem. It can be used to find the best approximation of the original data using fewer dimensions. Unlike PCA, SVD works on the original data matrix.



SVD does not need a covariance or correlation matrix. It works on the original data matrix.



SVD factors an $m \times n$ matrix A into a product of three matrices:

$$A = U * S * V^T$$

Here, U is an $m \times k$ matrix, V is an $n \times k$ matrix, and S is a $k \times k$ matrix. The columns of U are called left singular vectors and columns of V are called right singular vectors.

The values on the diagonal of the S matrix are called singular values.

Getting ready

We will use the Iris dataset for this exercise. Our task in hand is to reduce the dimensionality of the dataset from four to two.

How to do it...

Let's load the necessary libraries and get the Iris dataset:

```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import scale
from scipy.linalg import svd

# Load Iris dataset
data = load_iris()
x = data['data']
y = data['target']

# Proceed by scaling the x variable w.r.t its mean,
x_s = scale(x,with_mean=True,with_std=False,axis=0)

# Decompose the matrix using SVD technique. We will use SVD
implementation in scipy.
U,S,V = svd(x_s,full_matrices=False)

# Approximate the original matrix by selecting only the first two
singular values.
```

```
x_t = U[:, :2]

# Finally we plot the datasets with the reduced components.
plt.figure(1)
plt.scatter(x_t[:, 0], x_t[:, 1], c=y)
plt.xlabel("Component 1")
plt.ylabel("Component 2")
plt.show()
```

Now, we will demonstrate how to perform an SVD operation on the Iris dataset:

```
# Proceed by scaling the x variable w.r.t its mean,
x_s = scale(x, with_mean=True, with_std=False, axis=0)
# Decompose the matrix using SVD technique. We will use SVD
implementation in scipy.
U, S, V = svd(x_s, full_matrices=False)

# Approximate the original matrix by selecting only the first two
singular values.
x_t = U[:, :2]
```

```
# Finally we plot the datasets with the reduced components.
plt.figure(1)
plt.scatter(x_t[:, 0], x_t[:, 1], c=y)
plt.xlabel("Component 1")
plt.ylabel("Component 2")
plt.show()
```

How it works...

The Iris dataset has four columns. Though there are not many columns, it will serve our purpose. We intend to reduce the dimensionality of the Iris dataset to two from four and still retain all the information about the data.

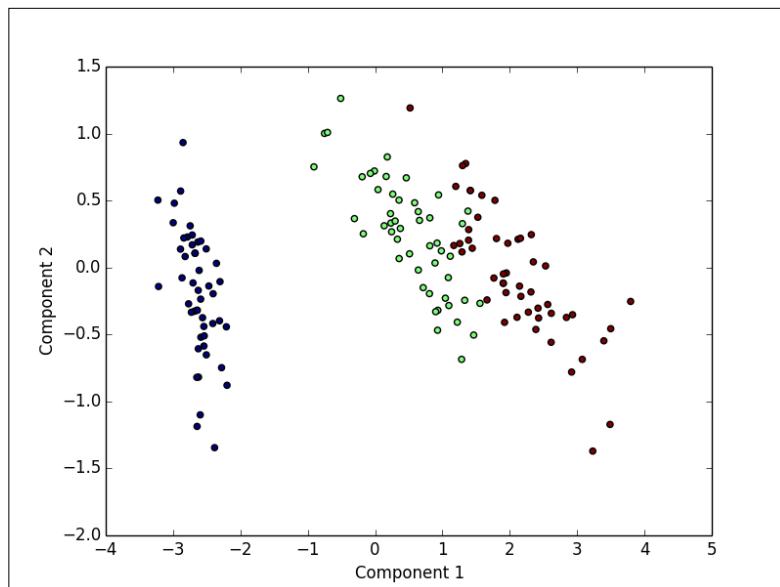
We will load the Iris data to the `x` and `y` variables using the convenient `load_iris` function from scikit-learn. The `x` variable is our data matrix; we can inspect its shape in the following manner:

```
>>>x.shape
(150, 4)
>>>
```

We center the data matrix x using its mean. The rule of thumb is that if all your columns are measured in the same scale and have the same unit of measurement in the data, you don't have to scale the data. This will allow PCA to capture these basis units with the maximum variation. Note that we used only the mean while invoking the function `scale`:

```
x_s = scale(x, with_mean=True, with_std=False, axis=0)
```

1. Run the SVD method on our scaled input dataset.
2. Select the top two singular components. This matrix is a reduced approximation of the original input data.
3. Finally, plot the columns and color it by the class value:



There's more...

SVD is a two-mode factor analysis, where we start with an arbitrary rectangular matrix with two types of entities. This is different from our previous recipe where we saw PCA that took a correlation matrix as an input. PCA is a one-mode factor analysis as the rows and columns in the input square matrix represent the same entity.

In text mining applications, the input is typically presented as a **Term-document Matrix (TDM)**. In a TDM, the rows correspond to the words and columns are the documents. The cell entries are filled with either the term frequency or **Term Frequency Inverse Document Frequency (TFIDF)** score. It is a rectangular matrix with two entities: words and documents that are present in the rows and columns of the matrix.

SVD is widely used in text mining applications to uncover the hidden relationships (semantic relationship) between words and documents, documents and documents, and words and words.

By applying SVD on a term-document matrix, we transform it into a new semantic space, where the words that do not occur together in the same document can still be close in the new semantic space. The goal of SVD is to find a useful way to model the relationship between the words and documents. After applying SVD, each document and word can be represented as a vector of the factor values. We can choose to ignore the components with very low values and, hence, avoid noises in the underlying dataset. This leads to an approximate representation of our text corpus. This is called **Latent Semantic Analysis (LSA)**.

The ramifications of this idea have a very high applicability in document indexing for search and information retrieval. Instead of indexing the original words as an inverted index, we can now index the output of LSA. This helps avoid problems such as synonymy and polysemy. In synonymy, users may tend to use different words to represent the same entity. A normal indexing is vulnerable to such scenarios. As the underlying document is indexed by regular words, a search may not yield the results. For example, if we indexed some documents related to financial instruments, typically the words would be currency, money, and similar stuff. Currency and money are synonymous words. While a user searches for money, he should be shown the documents related to currency as well. However, with regular indexing, the search engine would be able to retrieve only the documents having money. With latent semantic indexing, the documents with currency will also be retrieved. In the latent semantic space, currency and money will be close to each other as their neighboring words would be similar in the documents.

Polysemy is about words that have more than one meaning. For example, bank can refer to a financial institution or a river bank. Similar to synonymy, polysemy can also be handled in the latent semantic space.

For more information on LSA and latent semantic indexing, refer to the paper by Deerwester et al at:

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.108.8490>. For a comparative study of Eigen Values and Singular Values refer to the book Numerical Computing with MATLAB by Cleve Moler. Though the examples are in MATLAB, with the help of our recipe you can redo them in Python:

<https://in.mathworks.com/moler/eigs.pdf>

Reducing the data dimension with random projection

The methods that we saw previously for dimensionality reduction are computationally expensive and not the fastest ones. Random projection is another way to perform dimensionality reduction faster than these methods.

Random projections are attributed to the Johnson-Lindenstrauss lemma. According to the lemma, a mapping from a high-dimensional to a low-dimensional Euclidean space exists; such that the distance between the points is preserved within an epsilon variance. The goal is to preserve the pairwise distances between any two points in your data, and still reduce the number of dimensions in the data.

Let's say that if we are given n -dimensional data in any Euclidean space, according to the lemma, we can map it an Euclidean space of dimension k , such that all the distances between the points are preserved up to a multiplicative factor of $(1-\epsilon)$ and $(1+\epsilon)$.

Getting ready

For this exercise, we will use the 20 newsgroup data (<http://qwone.com/~jason/20Newsgroups/>).

It is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different categories of news. Scikit-learn provides a convenient function to load this dataset:

```
from sklearn.datasets import fetch_20newsgroups  
data = fetch_20newsgroups(categories='sci.crypt')
```

You can load all libraries or a list of categories of interest by providing a list of strings of categories. In our case, we will use the `sci.crypt` category.

We will load the input text as a term-document matrix where the features are individual words. On this, we will apply random projection in order to reduce the number of dimensions. We will try to see if the distances between the documents are preserved in the reduced space and an instance is a document.

How to do it...

Let's start with loading the necessary libraries. Using scikit's utility function `fetch20newsgroups`, we will load the data. We will select only the `sci.crypt` category out of all the data. We will then transform our text data into a vector representation:

```
from sklearn.datasets import fetch_20newsgroups  
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.metrics import euclidean_distances
from sklearn.random_projection import GaussianRandomProjection
import matplotlib.pyplot as plt
import numpy as np

# Load 20 newsgroup dataset
# We select only sci.crypt category
# Other categories include
# 'sci.med', 'sci.space' , 'soc.religion.christian'
cat =['sci.crypt']
data = fetch_20newsgroups(categories=cat)

# Create a term document matrix, with term frequencies as the values
# from the above dataset.
vectorizer = TfidfVectorizer(use_idf=False)
vector = vectorizer.fit_transform(data.data)

# Perform the projection. In this case we reduce the dimension to 1000
gauss_proj = GaussianRandomProjection(n_components=1000)
gauss_proj.fit(vector)
# Transform the original data to the new space
vector_t = gauss_proj.transform(vector)

# print transformed vector shape
print vector.shape
print vector_t.shape

# To validate if the transformation has preserved the distance, we
# calculate the old and the new distance between the points
org_dist = euclidean_distances(vector)
red_dist = euclidean_distances(vector_t)

diff_dist = abs(org_dist - red_dist)

# We take the difference between these points and plot them
# as a heatmap (only the first 100 documents).
plt.figure()
plt.pcolor(diff_dist[0:100,0:100])
plt.colorbar()
plt.show()
```

Let's now proceed to demonstrate the concept of random projection.

How it works...

After loading the newsgroup dataset, we will convert it to a matrix through `TfidfVectorizer(use_idf=False)`.

Notice that we have set `use_idf` to `False`. This creates our input matrix where the rows are documents, columns are individual words, and cell values are word counts.

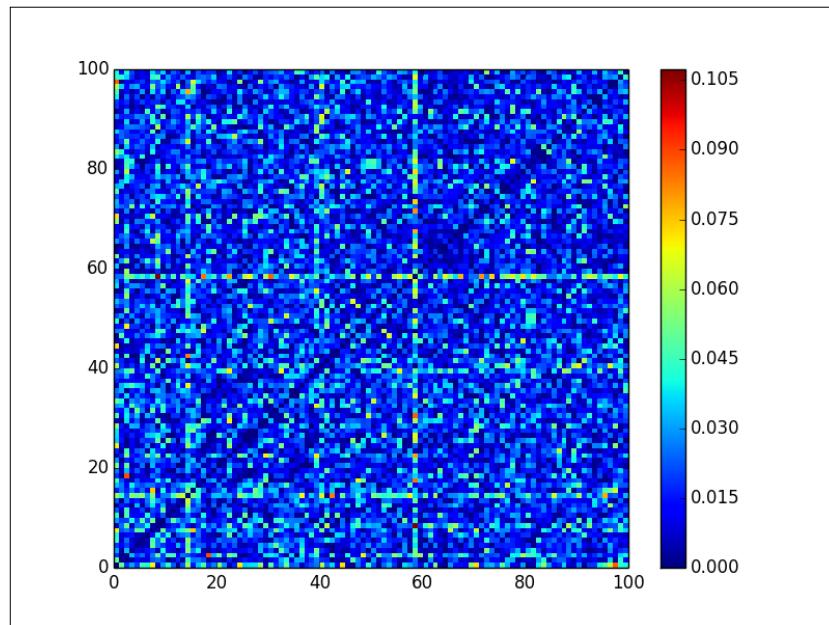
If we print our vector using the `print vector.shape` command, we will get the following output:

```
(595, 16115)
```

We can see that our input matrix has 595 documents and 16115 words; each word is a feature and, hence, a dimension.

We will perform the projection of the data using a dense Gaussian matrix. The Gaussian random matrix is generated by sampling elements from a normal distribution $N(0, 1/\text{number of components})$. In our case, the number of components is 1000. Our intention is to reduce the dimension to 1000 from 16115. We will then print the original and the reduced dimension in order to verify the reduction in the dimension.

Finally, we would like to validate whether the data characteristics are maintained after the projection. We will calculate the Euclidean distances between the vectors. We will record the distances in the original space and in the projected space as well. We will take a difference between them as in step 7 and plot the difference as a heat map:



As you can see, the gradient is in the range of 0.000 to 0.105 and indicates the difference in distance of vectors in the original and reduced space. The difference between the distance in the original space and projected space are pretty much in a very small range.

There's more...

There are a lot of references for random projections. It is a very active field of research. Interested readers can refer to the following papers:

Experiments with random projections:

<http://dl.acm.org/citation.cfm?id=719759>

Experiments with random projections for machine learning:

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.9205>

In our recipe, we used the Gaussian random projection where a Gaussian random matrix was generated by sampling from a normal distribution, $N(0,1/1000)$, where 1000 is the required dimension of the reduced space.

However, having a dense matrix can create severe memory-related issues while processing. In order to avoid this, Achlioptas proposed sparse random projections. Instead of choosing from a standard normal distribution, the entries are picked from $\{-1,0,1\}$ with a probability of $\{1/6, 2/3, 1/6\}$. As you can see, the probability of having 0 is two-thirds and, hence, the resultant matrix will be sparse. Users can refer to the seminal paper by Achlioptas at *Dimitris Achlioptas, Database-friendly random projections: Johnson-Lindenstrauss with binary coins. Journal of Computer and System Sciences, 66(4):671–687, 2003.*

The scikit implementation allows the users to choose the density of the resultant matrix. Let's say that if we specify the density as d and s as $1/d$, then the elements of the matrix are picked from the following equation:

$$-\sqrt{\frac{d}{No\ components}}, 0, +\sqrt{\frac{d}{No\ components}}$$

With probability of the following:

$$\left\{ \frac{1}{2s}, 1 - \frac{1}{s}, \frac{1}{2s} \right\}$$

See also

- ▶ Using kernel PCA recipe in Chapter 4, Analyzing Data - Deep Dive

Decomposing the feature matrices using non-negative matrix factorization

We discussed all the previous matrix decomposition recipes from a data dimensionality reduction perspective. Let's discuss this recipe from a collaborative filtering perspective to make it more interesting. Though data dimensionality reduction is what we are after, **Non-negative Matrix Factorization (NMF)** is used extensively in recommendation systems using a collaborative filtering algorithm.

Let's say that our input matrix A is of a dimension $m \times n$. NMF factorizes the input matrix into two matrices, A_{dash} and H :

$$A = A_{\text{dash}} * H$$

Let's say that we want to reduce the dimension of the A matrix to d , that is, we want the original $m \times n$ matrix to be decomposed into $m \times d$, where $d \ll n$.

The A_{dash} matrix is of a size $m \times d$ and the H matrix is of a size $d \times m$. NMF solves this as an optimization problem, that is, minimizing the function:

$$|A - A_{\text{dash}} * H|^2$$

The famous Netflix challenge was solved using NMF. Please refer to the following link:

Gábor Takács et al., (2008). *Matrix factorization and neighbor based algorithms for the Netflix prize problem*. In: *Proceedings of the 2008 ACM Conference on Recommender Systems, Lausanne, Switzerland, October 23 - 25, 267-274*:

<http://dl.acm.org/citation.cfm?id=1454049>

Getting ready

In order to explain NMF, let's create a toy recommendation problem. In a typical recommendation system such as the one with MovieLens or Netflix, there is a group of users and group of items (movies). If each user has rated a few movies, we want to predict their ratings for the movies that they have not rated. We will assume that the users have not watched the movies that they have not rated. Our prediction algorithm output is the ratings for these movies. We can then recommend the movies that have a very high rating from our prediction engine to these users.

Our toy problem is set as follows; we have the following movies:

Movie ID	Movie Name
1	Star Wars
2	The Matrix
3	Inception
4	Harry Potter
5	The Hobbit
6	Guns of Navarone
7	Saving Private Ryan
8	Enemy at the Gates
9	Where Eagles Dare
10	The Great Escape

We have ten movies, each identified with a movie ID. We also have 10 users who have rated these movies as follows:

	Movie ID									
User ID	1	2	3	4	5	6	7	8	9	10
1	5.0	5.0	4.5	4.5	5.0	3.0	2.0	2.0	0.0	0.0
2	4.2	4.7	5.0	3.7	3.5	0.0	2.7	2.0	1.9	0.0
3	2.5	0.0	3.3	3.4	2.2	4.6	4.0	4.7	4.2	3.6
4	3.8	4.1	4.6	4.5	4.7	2.2	3.5	3.0	2.2	0.0
5	2.1	2.6	0.0	2.1	0.0	3.8	4.8	4.1	4.3	4.7
6	4.7	4.5	0.0	4.4	4.1	3.5	3.1	3.4	3.1	2.5
7	2.8	2.4	2.1	3.3	3.4	3.8	4.4	4.9	4.0	4.3
8	4.5	4.7	4.7	4.5	4.9	0.0	2.9	2.9	2.5	2.1
9	0.0	3.3	2.9	3.6	3.1	4.0	4.2	0.0	4.5	4.6
10	4.1	3.6	3.7	4.6	4.0	2.6	1.9	3.0	3.6	0.0

For readability, we have kept it as a matrix where the rows correspond to the users and columns correspond to the movies. The cell values are the ratings from 1 to 5, where 5 signifies a high user affinity to the movie and 1 signifies the user's dislike. There are 0 values in the cells that indicate that the user has not rated those movies. In this recipe, we will decompose the user_id x movie_id matrix using NMF.

How to do it...

We will start with loading the necessary libraries and then creating our dataset. We will store our dataset as a matrix:

```
import numpy as np
from collections import defaultdict
from sklearn.decomposition import NMF
import matplotlib.pyplot as plt

# load our ratings matrix in python.
ratings = [
[5.0, 5.0, 4.5, 4.5, 5.0, 3.0, 2.0, 2.0, 0.0, 0.0],
[4.2, 4.7, 5.0, 3.7, 3.5, 0.0, 2.7, 2.0, 1.9, 0.0],
[2.5, 0.0, 3.3, 3.4, 2.2, 4.6, 4.0, 4.7, 4.2, 3.6],
[3.8, 4.1, 4.6, 4.5, 4.7, 2.2, 3.5, 3.0, 2.2, 0.0],
[2.1, 2.6, 0.0, 2.1, 0.0, 3.8, 4.8, 4.1, 4.3, 4.7],
[4.7, 4.5, 0.0, 4.4, 4.1, 3.5, 3.1, 3.4, 3.1, 2.5],
[2.8, 2.4, 2.1, 3.3, 3.4, 3.8, 4.4, 4.9, 4.0, 4.3],
[4.5, 4.7, 4.7, 4.5, 4.9, 0.0, 2.9, 2.9, 2.5, 2.1],
[0.0, 3.3, 2.9, 3.6, 3.1, 4.0, 4.2, 0.0, 4.5, 4.6],
[4.1, 3.6, 3.7, 4.6, 4.0, 2.6, 1.9, 3.0, 3.6, 0.0]
]

movie_dict = {
1:"Star Wars",
2:"Matrix",
3:"Inception",
4:"Harry Potter",
5:"The hobbit",
6:"Guns of Navarone",
7:"Saving Private Ryan",
8:"Enemy at the gates",
9:"Where eagles dare",
10:"Great Escape"
}
```

```
A = np.asmatrix(ratings,dtype=float)

# perform non negative matrix transformation on the data.
max_components = 2
reconstruction_error = []
nmf = None
nmf = NMF(n_components = max_components,random_state=1)
A_dash = nmf.fit_transform(A)

# Examine the reduced matrixfor i in range(A_dash.shape[0]):
for i in range(A_dash.shape[0]):
    print "User id = %d, comp1 score = %0.2f, comp 2 score =
%0.2f"%(i+1,A_dash[i][0],A_dash[i][1])

plt.figure(1)
plt.title("User Concept Mapping")
x = A_dash[:,0]
y = A_dash[:,1]
plt.scatter(x,y)
plt.xlabel("Component 1 Score")
plt.ylabel("Component 2 Score")

# Let us examine our component matrix F.
F = nmf.components_
plt.figure(2)
plt.title("Movie Concept Mapping")
x = F[0,:]
y = F[1,:]
plt.scatter(x,y)
plt.xlabel("Component 1 score")
plt.ylabel("Component 2 score")
for i in range(F[0,:].shape[0]):
    plt.annotate(movie_dict[i+1],(F[0,:][i],F[1,:][i]))
plt.show()
```

Let's now proceed to demonstrate a non-negative matrix transformation:

```
# perform non negative matrix transformation on the data.
max_components = 2
reconstruction_error = []
nmf = None
nmf = NMF(n_components = max_components,random_state=1)
A_dash = nmf.fit_transform(A)

# Examine the reduced matrixfor i in range(A_dash.shape[0]):
for i in range(A_dash.shape[0]):
    print "User id = %d, comp1 score = %0.2f, comp 2 score =
%0.2f"%(i+1,A_dash[i][0],A_dash[i][1])
```

```
plt.figure(1)
plt.title("User Concept Mapping")
x = A_dash[:,0]
y = A_dash[:,1]
plt.scatter(x,y)
plt.xlabel("Component 1 Score")
plt.ylabel("Component 2 Score")

# Let us examine our component matrix F.
F = nmf.components_
plt.figure(2)
plt.title("Movie Concept Mapping")
x = F[0,:]
y = F[1,:]
plt.scatter(x,y)
plt.xlabel("Component 1 score")
plt.ylabel("Component 2 score")
for i in range(F[0,:].shape[0]):
    plt.annotate(movie_dict[i+1], (F[0,:] [i], F[1,:] [i]))
plt.show()
```

How it works...

We will load the data to a NumPy matrix A from the list. We will choose to reduce the dimension to two as dictated by the `max_components` variable. We will initialize the NMF object with the number of components. Finally, we will apply the algorithm to get the reduced matrix, `A_dash`.

That's all we need to do. The scikit library hides a lot of details for us. Let's now look at what is happening in the background. Formally, NMF decomposes the original matrix into two matrices, which when multiplied together, give the approximation of our original matrix. Look at the following line in our code:

```
A_dash = nmf.fit_transform(A)
```

The input matrix A is transformed into a reduced matrix, `A_dash`. Let's look at the shape of the new matrix:

```
>>>A_dash.shape
(10, 2)
```

The original matrix is reduced to two columns as compared to the original ten columns. This is the reduced space. From this data perspective, we can say that our algorithm has now grouped our original ten movies into two concepts. The cell value indicates the user affinity towards each of the concepts.

We will print and see how the affinity looks:

```
for i in range(A_dash.shape[0]):  
    print User id = %d, comp1 score = %0.2f, comp 2 score =%0.2f%(i+1,A_  
dash[i][0],A_dash[i][1])
```

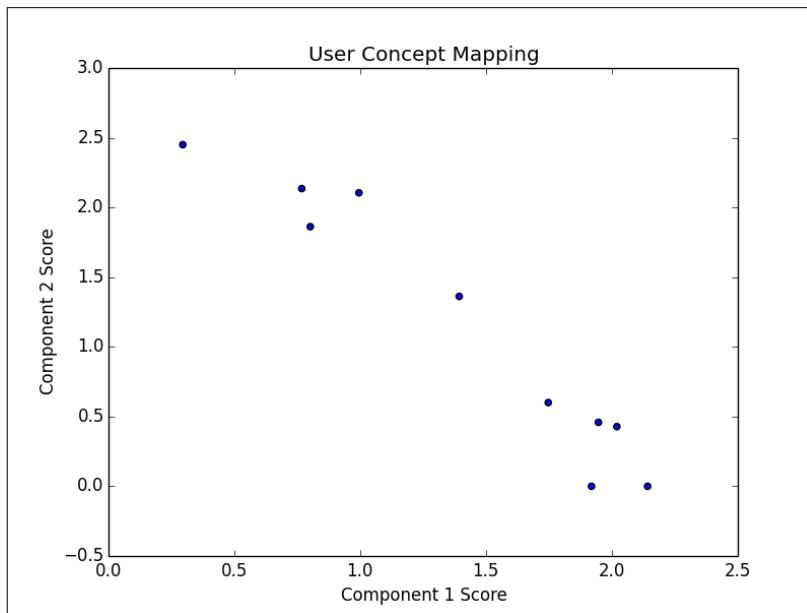
The output looks as follows:

```
User id = 1, comp1 score = 2.14, comp 2 score = 0.00  
User id = 2, comp1 score = 1.92, comp 2 score = 0.00  
User id = 3, comp1 score = 0.77, comp 2 score = 2.14  
User id = 4, comp1 score = 1.95, comp 2 score = 0.46  
User id = 5, comp1 score = 0.30, comp 2 score = 2.45  
User id = 6, comp1 score = 1.39, comp 2 score = 1.36  
User id = 7, comp1 score = 0.99, comp 2 score = 2.10  
User id = 8, comp1 score = 2.02, comp 2 score = 0.43  
User id = 9, comp1 score = 0.80, comp 2 score = 1.86  
User id = 10, comp1 score = 1.75, comp 2 score = 0.60
```

Let's look at user 1; the first line in the preceding image says that user 1 has a score of 2.14 for concept 1 and 0 for concept 2, indicating that user 1 has more affinity towards concept 1.

Look at user ID 3; this user has more affinity towards concept 1. Now we have reduced our input dataset to two dimensions, it would be nice to view this in a graph.

In our x axis, we have component 1 and our y axis is component 2. We will plot the various users as a scatter plot. Our graph looks as follows:



You can see that we have two groups of users; all those with a component 1 score of greater than 1.5 and others with less than 1.5. We are able to group our users into two clusters in the reduced feature space.

Let's look at the other matrix, F:

```
F = nmf.components_
```

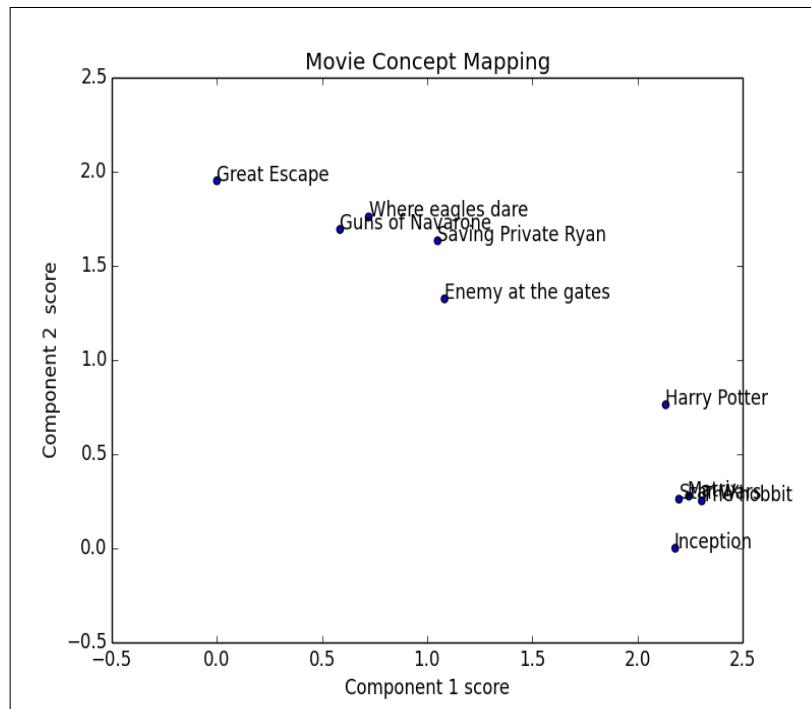
The F matrix has two rows; each row corresponds to our component and ten columns, each corresponding to a movie ID. Another way to look at it is the affinity of movies towards these concepts. Let's plot the matrix.

You can see that our x axis is the first row and the y axis is the second row. In step 1, we declared a dictionary. We want this dictionary to annotate each point with the movie name:

```
for i in range(F[0, :].shape[0]):  
    plt.annotate(movie_dict[i+1], (F[0, :] [i], F[1, :] [i]))
```

The `annotate` method takes the string (used to annotate) as the first parameter and the x and y coordinates as a tuple.

You can see the output graph as follows:



You can see that we have two distinct groups. All the war movies have a very low component 1 score and a very high component 2 score. All the fantasy movies have a vice versa score. We can safely say that component 1 comprises of war movies and the users having high component 1 scores have very high affinity towards war movies. The same can be said for fantasy movies.

Thus, using NMF, we are able to unearth the hidden features in our input matrix with respect to the movies.

There's more...

We saw how the feature space was reduced from ten dimensions to two dimensions for the users. Now, let's see how this can be used for the recommendation engines. Let's reconstruct the original matrix from the two matrices:

```
reconstructed_A = np.dot(W,H)
np.set_printoptions(precision=1)
print reconstructed_A
```

The reconstructed matrix looks as follows:

[[4.7 4.8 4.7 4.6 4.9 1.3 2.2 2.3 1.5 0.]
[4.2 4.3 4.2 4.1 4.4 1.1 2. 2.1 1.4 0.]
[2.2 2.3 1.7 3.3 2.3 4.1 4.3 3.7 4.3 4.2]
[4.4 4.5 4.2 4.5 4.6 1.9 2.8 2.7 2.2 0.9]
[1.3 1.3 0.6 2.5 1.3 4.3 4.3 3.6 4.5 4.8]
[3.4 3.5 3. 4. 3.6 3.1 3.7 3.3 3.4 2.7]
[2.7 2.8 2.2 3.7 2.8 4.1 4.5 3.9 4.4 4.1]
[4.6 4.7 4.4 4.6 4.8 1.9 2.8 2.8 2.2 0.8]
[2.2 2.3 1.7 3.1 2.3 3.6 3.9 3.3 3.9 3.6]
[4. 4.1 3.8 4.2 4.2 2. 2.8 2.7 2.3 1.2]]]

How different is it from the original matrix? The original matrix is given here; look at the highlighted row:

	Movie ID									
User ID	1	2	3	4	5	6	7	8	9	10
1	5.0	5.0	4.5	4.5	5.0	3.0	2.0	2.0	0.0	0.0
2	4.2	4.7	5.0	3.7	3.5	0.0	2.7	2.0	1.9	0.0
3	2.5	0.0	3.3	3.4	2.2	4.6	4.0	4.7	4.2	3.6
4	3.8	4.1	4.6	4.5	4.7	2.2	3.5	3.0	2.2	0.0
5	2.1	2.6	0.0	2.1	0.0	3.8	4.8	4.1	4.3	4.7

	Movie ID										
6	4.7	4.5	0.0	4.4	4.1	3.5	3.1	3.4	3.1	2.5	
7	2.8	2.4	2.1	3.3	3.4	3.8	4.4	4.9	4.0	4.3	
8	4.5	4.7	4.7	4.5	4.9	0.0	2.9	2.9	2.5	2.1	
9	0.0	3.3	2.9	3.6	3.1	4.0	4.2	0.0	4.5	4.6	
10	4.1	3.6	3.7	4.6	4.0	2.6	1.9	3.0	3.6	0.0	

For user 6 and movie 3, we now have a rating. This will help us decide whether to recommend this movie to the user, as he has not watched it. Remember that this is a toy dataset; real-world scenarios have many movies and users.

See also

- ▶ *Extracting Features Using Singular Value Decomposition* recipe in Chapter 4,
Analyzing Data - Deep Dive

5

Data Mining – Needle in a Haystack

In this chapter, we will cover the following topics:

- ▶ Working with distance measures
- ▶ Learning and using kernel methods
- ▶ Clustering data using the k-means method
- ▶ Learning vector quantization
- ▶ Finding outliers in univariate data
- ▶ Discovering outliers using the local outlier factor method

Introduction

In this chapter, we will focus mostly on unsupervised data mining algorithms. We will start with a recipe covering various distance measures. Understanding distance measures and various spaces is critical when building data science applications. Any dataset is usually a set of points that are objects belonging to a particular space. We can define space as a universal set of points from which the points in our dataset are drawn. The most often encountered space is Euclidean. In Euclidean space, the points are vectors real number. The length of the vector denotes the number of dimensions.

We then have a recipe introducing kernel methods. Kernel methods are a very important topic in machine learning. They help us solve nonlinear data problems using linear methods. We will introduce the concept of the kernel trick.

We will follow it with some clustering algorithm recipes. Clustering is the process of partitioning a set of points into logical groups. For example, in a supermarket scenario, items are grouped into categories qualitatively. However, we will look at quantitative approaches. Specifically, we will focus our attention on the k-means algorithm and discuss its limitations and advantages.

Our next recipe is an unsupervised technique called learning vector quantization. It can be used both for clustering and classification tasks.

Finally, we will look at the outlier detection methods. Outliers are those observations in a dataset that differ significantly from the other observations in that dataset. It is very important to study these outliers as they might be indicative of unusual phenomena or errors in the underlying process that is generating the data. When machine learning models are fitted over data, it is important to understand how to handle outliers before passing the data to algorithms. We will concentrate on a few empirical outlier detection techniques in this chapter.

We will rely heavily on the Python libraries, NumPy, SciPy, matplotlib, and scikit-learn for most of our recipes. We will also change our coding style from scripting to writing procedures and classes in this chapter.

Working with distance measures

Distance and similarity measures are key to various data mining tasks. In this recipe, we will see some distance measures in action. Our next recipe will cover similarity measures. Let's define a distance measure before we look at the various distance metrics.

As data scientists, we are always presented with points or vectors of different dimensions. Mathematically, a set of points is defined as a space. A distance measure in this space is defined as a function $d(x,y)$, which takes two points x and y as arguments in this space and gives a real number as the output. The distance function, that is, the real number output, should satisfy the following axioms:

1. The distance function output should be non-negative, $d(x,y) \geq 0$
2. The output of the distance function should be zero only when $x = y$
3. The distance should be symmetric, that is, $d(x,y) = d(y,x)$
4. The distance should obey the triangle inequality, that is, $d(x,y) \leq d(x,z) + d(z,y)$

A careful look at the fourth axiom reveals that distance is the length of the shortest path between two points.

You can refer to the following link for more information on the axioms:

http://en.wikipedia.org/wiki/Metric_%28mathematics%29

Getting ready

We will look at distance measures in Euclidean and non-Euclidean spaces. We will start with Euclidean distance and then define L_r-norm distance. L_r-norm is a family of distance measures of which Euclidean is a member. We will then follow it with the cosine distance. In non-Euclidean spaces, we will look at Jaccard's distance and Hamming distance.

How to do it...

Let's start by defining the functions to calculate the various distance measures:

```
import numpy as np

def euclidean_distance(x,y):
    if len(x) == len(y):
        return np.sqrt(np.sum(np.power((x-y),2)))
    else:
        print "Input should be of equal length"
    return None


def lrNorm_distance(x,y,power):
    if len(x) == len(y):
        return np.power(np.sum(np.power((x-y),power)),(1/(1.0*power)))
    else:
        print "Input should be of equal length"
    return None


def cosine_distance(x,y):
    if len(x) == len(y):
        return np.dot(x,y) / np.sqrt(np.dot(x,x) * np.dot(y,y))
    else:
        print "Input should be of equal length"
    return None


def jaccard_distance(x,y):
    set_x = set(x)
    set_y = set(y)
    return 1 - len(set_x.intersection(set_y)) / len(set_x.union(set_y))
```

```
def hamming_distance(x,y):  
    diff = 0  
    if len(x) == len(y):  
        for char1,char2 in zip(x,y):  
            if char1 != char2:  
                diff+=1  
    return diff  
else:  
    print "Input should be of equal length"  
return None
```

Now, let's write a main routine in order to invoke these various distance measure functions:

```
if __name__ == "__main__":  
  
    # Sample data, 2 vectors of dimension 3  
    x = np.asarray([1,2,3])  
    y = np.asarray([1,2,3])  
    # print euclidean distance  
    print euclidean_distance(x,y)  
    # Print euclidean by invoking lr norm with  
    # r value of 2  
    print lrNorm_distance(x,y,2)  
    # Manhattan or citi block Distance  
    print lrNorm_distance(x,y,1)  
  
    # Sample data for cosine distance  
    x =[1,1]  
    y =[1,0]  
    print 'cosine distance'  
    print cosine_distance(x,y)  
  
    # Sample data for jaccard distance  
    x = [1,2,3]  
    y = [1,2,3]  
    print jaccard_distance(x,y)  
  
    # Sample data for hamming distance  
    x = [11001]  
    y = [11011]  
    print hamming_distance(x,y)
```

How it works...

Let's look at the main function. We created a sample dataset and two vectors of three dimensions and invoked the `euclidean_distance` function.

This is the most common distance measure used is Euclidean distance. It belongs to a family of the L_r-Norm distance. A space is defined as a Euclidean space if the points in this space are vectors composed of real numbers. It's also called the L₂-norm distance. The formula for Euclidean distance is as follows:

$$d([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n]) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

As you can see, Euclidean distance is derived by finding the distance in each dimension (subtracting the corresponding dimensions), squaring the distance, and finally taking a square root.

In our code, we leverage NumPy square root and power function in order to implement the preceding formula:

```
np.sqrt(np.sum(np.power((x-y), 2)))
```

Euclidean distance is strictly positive. When `x` is equal to `y`, the distance is zero. This should become clear from how we invoked Euclidean distance:

```
x = np.asarray([1,2,3])
y = np.asarray([1,2,3])

print euclidean_distance(x,y)
```

As you can see, we defined two NumPy arrays, `x` and `y`. We have kept them the same. Now, when we invoke the `euclidean_distance` function with these parameters, our output is zero.

Let's now invoke the L₂-norm function, `lrNorm_distance`.

The L_r-Norm distance metric is from a family of distance metrics of which Euclidean distance is a member. This should become clear as we see its formula:

$$d([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n]) = \left(\sum_{i=1}^n |x_i - y_i|^r \right)^{\frac{1}{r}}$$

You can see that we now have a parameter, `r`. Let's substitute `r` with 2. This will turn the preceding equation to a Euclidean equation. Hence, Euclidean is called the L2-norm distance:

```
lrNorm_distance(x,y,power) :
```

In addition to two vectors, we will also pass a third parameter called `power`. This is the `r` defined in the formula. Invoking it with a power value set to two will yield the Euclidean distance. You can check it by running the following code:

```
print lrNorm_distance(x,y,2)
```

This will yield zero as a result, which is similar to the Euclidean distance function.

Let's define two sample vectors, `x` and `y`, and invoke the `cosine_distance` function.

In the spaces where the points are considered as directions, the cosine distance yields a cosine of the angle between the given input vectors as a distance value. Both the Euclidean space also the spaces where the points are vectors of integers or Boolean values, are candidate spaces where the cosine distance function can be applied. The cosine of the angle between the input vectors is the ratio of a dot product of the input vectors to the product of an L2-norm of individual input vectors:

```
np.dot(x,y) / np.sqrt(np.dot(x,x) * np.dot(y,y))
```

Let's look at the numerator where the dot product between the input vector is calculated:

```
np.dot(x,y)
```

We will use the NumPy dot function to get the dot product value. The dot product for the two vectors, `x` and `y`, is defined as follows:

$$\sum_{i=1}^n x_i * y_i$$

Now, let's look at the denominator:

```
np.sqrt(np.dot(x,x) * np.dot(y,y))
```

We again use the dot function to find the L2-norm of our input vectors:

```
np.dot(x,x) is equivalent to
```

```
tot = 0
for i in range(len(x)):
    tot+=x[i] * x[i]
```

Thus, we can calculate the cosine of the angle between the two input vectors.

We will move on to Jaccard's distance. Similar to the previous invocations, we will define the sample vectors and invoke the `jaccard_distance` function.

From vectors of real values, let's move on to sets. Commonly called Jaccard's coefficient, it is the ratio of the sizes of the intersection and the union of the given input vectors. One minus this value gives the Jaccard's distance. As you can see, in the implementation, we first converted the input lists to sets. This will allow us to leverage the union and intersection operations provided by the Python set datatype:

```
set_x = set(x)
set_y = set(y)
```

Finally, the distance is calculated as follows:

```
1 - len(set_x.intersection(set_y)) / (1.0 * len(set_x.union(set_y)))
```

We must use the intersection and union functionalities that are available in the `set` datatype in order to calculate the distance.

Our last distance metric is the Hamming distance. With two bit vectors, the Hamming distance calculates how many bits have differed in these two vectors:

```
for char1,char2 in zip(x,y):
    if char1 != char2:
        diff+=1
return diff
```

As you can see, we used the `zip` functionality to check each of the bits and maintain a counter on how many bits have differed. The Hamming distance is used with a categorical variable.

There's more...

Remember that by subtracting one from our distance values, we can arrive at a similarity value.

Yet another distance that we didn't go into in detail, but is used prevalently, is the Manhattan or city block distance. It's an L1-norm distance. By passing an `r` value as `1` to the `Lr-norm` distance function, we will get the Manhattan distance.

Depending on the underlying space in which the data is placed, an appropriate distance measure needs to be selected. When using these distances in algorithms, we need to be mindful about the underlying space. For example, in the k-means algorithm, at every step cluster center is calculated as an average of all the points that are close to each other. A nice property of Euclidean is that the average of the points exists and as a point in the same space. Note that our input for the Jaccard's distance was sets. An average of the sets does not make any sense.

While using the cosine distance, we need to check whether the underlying space is Euclidean or not. If the elements of the vectors are real numbers, then the space is Euclidean, if they are integers, then the space is non-Euclidean. The cosine distance is most commonly used in text mining. In text mining, the words are considered as the axes, and a document is a vector in this space. The cosine of the angle between two document vectors denotes how similar the two documents are.

SciPy has an implementation of all these distance measures listed and much more at:

<http://docs.scipy.org/doc/scipy/reference/spatial.distance.html>.

The above URL lists all the distance measures supported by SciPy.

Additionally, the scikit-learn `pairwise` submodule provides you with a method called `pairwise_distance`, which can be used to find out the distance matrix from input records. This can be found at:

http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

We had mentioned that the Hamming distance is used with a categorical variable. A point worth mentioning here is the one-hot encoding that is used typically for categorical variables. After the one-hot encoding, the Hamming distance can be used as a similarity/distance measure between the input vectors.

See also

- ▶ Reducing data dimension with Random Projections recipe in Chapter 4, Analyzing Data - Deep Dive

Learning and using kernel methods

In this recipe, we will learn how to use kernel methods for data processing. Having the knowledge of kernels in your arsenal of methods will help you in dealing with nonlinear problems. This recipe is an introduction to kernel methods.

Typically, linear models—models that can separate the data using a straight line or hyper plane—are easy to interpret and understand. Nonlinearity in the data stops us from using linear models effectively. If the data can be transformed into a space where the relationship becomes linear, we can use linear models. However, mathematical computation in the transformed space can turn into a costly operation. This is where the kernel functions come to our rescue.

Kernels are similarity functions. It takes two input parameters, and the similarity between the two inputs is the output of the kernel function. In this recipe, we will look at how kernel achieves this similarity. We will also discuss what is called a kernel trick.

Formally defining a kernel K is a similarity function: $K(x_1, x_2) > 0$ denotes the similarity of x_1 and x_2 .

Getting ready

Let's define it mathematically before looking at the various kernels:

$$k(x_i, j_i) = \langle \varphi(x_i), \varphi(x_j) \rangle$$

Here, x_i and, x_j are the input vectors:

$$\varphi(x_i), \varphi(x_j)$$

The above mapping function is used to transform the input vectors into a new space. For example, if the input vector is in an n-dimensional space, the transformation function transforms it into a new space of dimension, m, where $m \gg n$:

$$\langle \varphi(x_i), \varphi(x_j) \rangle$$



The above image denotes the dot product:

$$\langle \varphi(x_i), \varphi(x_j) \rangle$$

The above image is the dot product, x_i and x_j are now transformed into a new space by the mapping function.

In this recipe, we will see a simple kernel in action.

Our mapping function will be as follows:

$$\varphi(x_1, x_2, x_3) = (x_1^2, x_2^2, x_3^2, x_1x_2, x_1x_3, x_2x_1, x_2x_3, x_3x_1, x_3x_2)$$

When the original data is supplied to this mapping function, it transforms the input into the new space.

How to do it...

Let's create two input vectors and define the mapping function as described in the previous section:

```
import numpy as np
# Simple example to illustrate Kernel Function concept.
# 3 Dimensional input space
x = np.array([10,20,30])
y = np.array([8,9,10])

# Let us find a mapping function to transform this space
# phi(x1,x2,x3) = (x1x2,x1x3,x2x3,x1x1,x2x2,x3x3)
# this will transform the input space into 6 dimesions

def mapping_function(x):
    output_list = []
    for i in range(len(x)):
        output_list.append(x[i]*x[i])

    output_list.append(x[0]*x[1])
    output_list.append(x[0]*x[2])
    output_list.append(x[1]*x[0])
    output_list.append(x[1]*x[2])
    output_list.append(x[2]*x[1])
    output_list.append(x[2]*x[0])
    return np.array(output_list)
```

Now, let's look at the main routine to invoke the kernel transformation. In the main function, we will define a kernel function and pass the input variable to the function, and print the output:

$$k(x,y) = \langle x, y \rangle^2$$

```
if __name__ == "__main__":
    # Apply the mapping function
    tranf_x = mapping_function(x)
    tranf_y = mapping_function(y)
    # Print the output
    print tranf_x
    print np.dot(tranf_x,tranf_y)

    # Print the equivalent kernel functions
    # transformation output.
    output = np.power((np.dot(x,y)),2)
    print output
```

How it works...

Let's follow this program from our main function. We created two input vectors, `x` and `y`. Both the vectors are of three dimensions.

We then defined a mapping function. The mapping function uses the input vector values and transforms the input vector into a new space with an increased dimension. In this case, the number of the dimension is increased to nine from three.

Let's now apply a mapping function on these vectors in order to increase their dimension to nine.

If we print `trans_x`, we will get the following:

```
[100 400 900 200 300 200 600 600 300]
```

As you can see, we transformed our input, `x`, from three dimensions to a nine-dimensional vector.

Now, let's take the dot product in the transformed space and print its output.

The output is 313600, a scalar value.

Let's now recap: we first transformed our two input vectors into a higher dimensional space and then calculated the dot product in order to derive a scalar output.

What we did was a very costly operation of transforming our original three-dimensional vector to a nine-dimensional vector and then performing the dot product operation on it.

Instead, we can choose a kernel function, which can arrive at the same scalar output without explicitly transforming the original space into a new space.

Our new kernel is defined as follows:

$$k(x, y) = \langle x, y \rangle^2$$

With two inputs, `x` and `y`, this kernel computes the dot product of the vectors, and squares them.

After printing the output from the kernel, we get 313600.

We never did the transformation but still were able to get the same result as the dot product output in the transformed space. This is called the kernel trick.

There was no magic in choosing this kernel. By expanding the kernel, we can arrive at our mapping function. Refer to the following reference for the expansion details:

http://en.wikipedia.org/wiki/Polynomial_kernel.

There's more...

There are several types of kernels. Based on our data characteristics and algorithm needs, we need to choose the right kernel. Some of them are as follows:

Linear kernel: This is the simplest kind of kernel function. For two given inputs, it returns the dot product of the input:

$$K(x, y) = x^T y$$

Polynomial kernel: This is defined as follows:

$$K(x, y) = (\gamma x^T y + c)^d$$

Here, x and y are the input vectors, d is the degree of the polynomial, and c is a constant. In our recipe, we used a polynomial kernel of degree 2.

The following is the scikit implementation of the linear and polynomial kernels:

http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.linear_kernel.html#sklearn.metrics.pairwise.linear_kernel

http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.polynomial_kernel.html#sklearn.metrics.pairwise.polynomial_kernel.

See also

- ▶ *Using Kernel PCA recipe in Chapter 4, Analyzing Data - Deep Dive*
- ▶ *Reducing data dimension with Random Projections recipe in Chapter 4, Analyzing Data - Deep Dive*

Clustering data using the k-means method

In this recipe, we will look at the k-means algorithm. K-means is a center-seeking unsupervised algorithm. It is an iterative non-deterministic method. What we mean by iterative is that the algorithm steps are repeated till the convergence of a specified number of steps.

Non-deterministic means that a different starting value may lead to a different final cluster assignment. The algorithm requires the number of clusters, k , as input. There is no good way to select the value of k , it has to be determined by running the algorithm multiple times.

For any clustering algorithm, the quality of its output is determined by inter-cluster cohesiveness and intra-cluster separation. Points in the same cluster should be close to each other; points in different clusters should be far away from each other.

Getting ready

Before we jump into how to write the k-means algorithm in Python, there are two key concepts that we need to cover that will help us understand better the quality of the output produced by our algorithm. First is a definition with respect to the quality of the clusters formed, and second is a metric that is used to find the quality of the clusters.

Every cluster detected by k-means can be evaluated using the following measures:

1. **Cluster location:** This is the coordinates of the cluster center. K-means starts with some random points as the cluster center and iteratively finds a new center around which points that are similar are grouped.
2. **Cluster radius:** This is the average deviation of all the points from the cluster center.
3. **Mass of the cluster:** This is the number of points in a cluster.
4. **Density of the cluster:** This is the ratio of mass of the cluster to its radius.

Now, we will measure the quality of our output clusters. As mentioned previously, this is an unsupervised problem and we don't have labels against which to check our output in order to get measures such as precision, recall, accuracy, F1-score, or other similar metrics. The metric that we will use for our k-means algorithm is called a silhouette coefficient. It takes values in the range of -1 to 1. Negative values indicate that the cluster radius is greater than the distance between the clusters so that the clusters overlap. This suggests poor clustering. Large values, that is, values close to 1, indicate good clustering.

A silhouette coefficient is defined for each point in the cluster. With a cluster, C, and a point, i , in this cluster, let x_i be the average distance of this point from all the other points in the cluster.

Now, calculate the average distance that the point i has from all the points in another cluster, D. Pick the smallest of these values and call it y_i :

$$S_i = \frac{y_i - x_i}{\max(x_i, y_i)}$$

For every cluster, the average of the silhouette coefficient of all the points can serve as a good measure of the cluster quality. An average of the silhouette coefficient of all the data points can serve as an overall quality metric for the clusters formed.

Let's go ahead and generate some random data:

```
import numpy as np
import matplotlib.pyplot as plt

def get_random_data():
    x_1 = np.random.normal(loc=0.2,scale=0.2,size=(100,100))
    x_2 = np.random.normal(loc=0.9,scale=0.1,size=(100,100))
    x = np.r_[x_1,x_2]
    return x
```

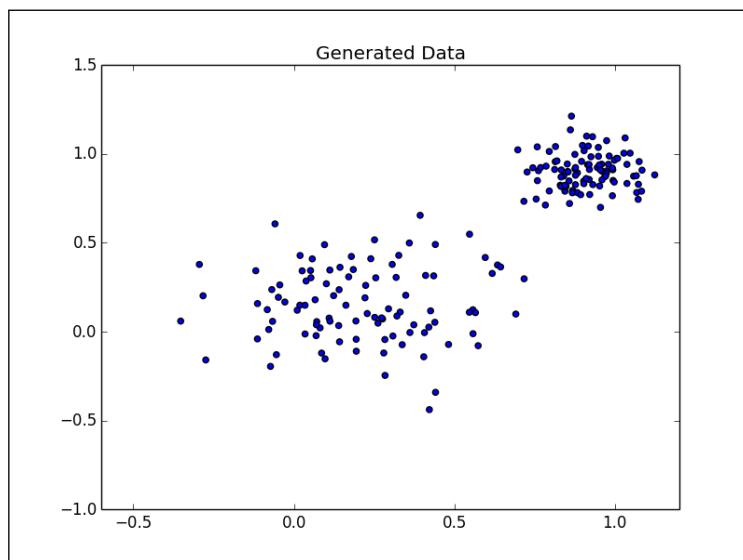
We sampled two sets of data from a normal distribution. The first set was picked up with a mean of 0.2 and standard deviation of 0.2. For the second set, our mean value was 0.9 and standard deviation was 0.1. Each dataset was a matrix of size 100 * 100—we have 100 instances and 100 dimensions. Finally, we merged both of them using the row stacking function from NumPy. Our final dataset was of size 200 * 100.

Let's do a scatter plot of the data:

```
x = get_random_data()

plt.cla()
plt.figure(1)
plt.title("Generated Data")
plt.scatter(x[:,0],x[:,1])
plt.show()
```

The plot is as follows:



Though we plotted only the first and second dimension, you can still clearly see that we have two clusters. Let's now jump into writing our k-means clustering algorithm.

How to do it...

Let's define a function that can perform the k-means clustering for the given data and a parameter, k. The function fits the clustering on the given data and returns an overall silhouette coefficient.

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

def form_clusters(x, k):
    """
    Build clusters
    """
    # k = required number of clusters
    no_clusters = k
    model = KMeans(n_clusters=no_clusters, init='random')
    model.fit(x)
    labels = model.labels_
    print labels
    # Calculate the silhouette score
    sh_score = silhouette_score(x, labels)
    return sh_score
```

Let's invoke the preceding function for the different values of k and store the returned silhouette coefficient:

```
sh_scores = []
for i in range(1,5):
    sh_score = form_clusters(x,i+1)
    sh_scores.append(sh_score)

no_clusters = [i+1 for i in range(1,5)]
```

Finally, let's plot the silhouette coefficient for the different values of k.

```
no_clusters = [i+1 for i in range(1,5)]

plt.figure(2)
plt.plot(no_clusters,sh_scores)
plt.title("Cluster Quality")
plt.xlabel("No of clusters k")
plt.ylabel("Silhouette Coefficient")
plt.show()
```

How it works...

As mentioned previously, k-means is an iterative algorithm. Roughly, the steps of k-means are as follows:

1. Initialize k random points from the dataset as initial center points.
2. Do the following till the convergence of the specified number of times:
 - ❑ Assign the points to the closest cluster center. Typically, Euclidean distance is used to find the distance between a point and the cluster center.
 - ❑ Recalculate the new cluster centers based on the assignment in this iteration.
 - ❑ Exit the loop if a cluster assignment of the points remains the same as the previous iteration. The algorithm has converged to an optimal solution.
3. We will leverage the k-means implementation from the scikit-learn library. Our cluster function takes the k value and dataset as a parameter and runs the k-means algorithm:

```
model = KMeans(n_clusters=no_clusters, init='random')
model.fit(x)
```

The no_clusters is the parameter that we will pass to the function. Using the init parameter, we set the initial center points as random. When init is set to random, scikit-learn estimates the mean and variance from the data and then samples k centers from a Gaussian distribution.

Finally, we must call the fit method to run k-means on our dataset:

```
labels = model.labels_
sh_score = silhouette_score(x, labels)
return sh_score
```

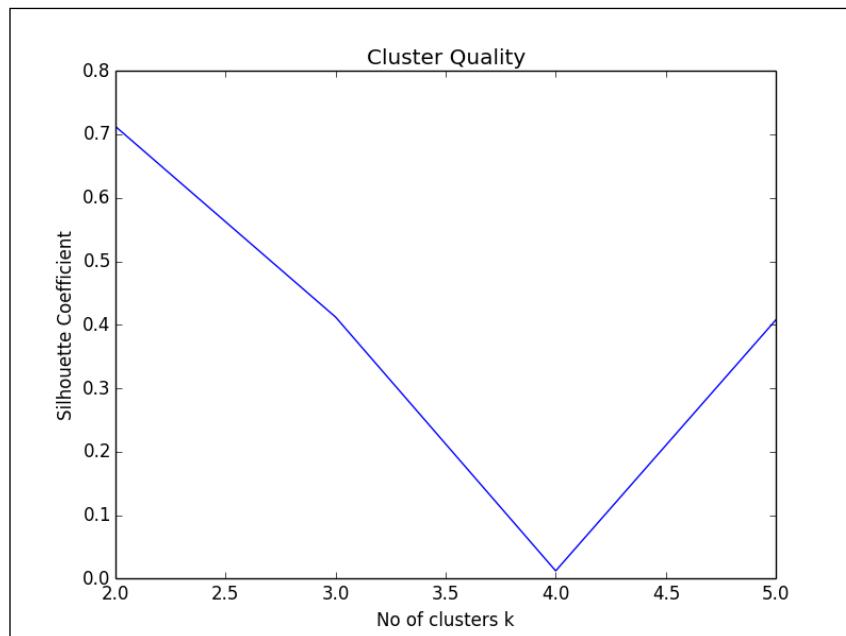
We get the labels, that is, the cluster assignment for each point and find out the silhouette coefficient for all the points in our cluster.

In real-world scenarios, when we start with the k-means algorithm on a dataset, we don't know the number of clusters present in the data; in other words, we don't know the ideal value for k. However, in our example, we know that k=2 as we generated the data in such a manner that it fits in two clusters. Hence, we need to run k-means for the different values of k:

```
sh_scores = []
for i in range(1,5):
    sh_score = form_clusters(x,i+1)
    sh_scores.append(sh_score)
```

For each run, that is, each value of k, we store the silhouette coefficient. A plot of k versus the silhouette coefficient reveals the ideal k value for the dataset:

```
no_clusters = [i+1 for i in range(1,5)]  
  
plt.figure(2)  
plt.plot(no_clusters,sh_scores)  
plt.title("Cluster Quality")  
plt.xlabel("No of clusters k")  
plt.ylabel("Silhouette Coefficient")  
plt.show()
```



As expected, our silhouette coefficient is very high for k=2.

There's more...

A couple of points to be noted about k-means. The k-means algorithm cannot be used for categorical data, k-medoids is used. Instead of averaging all the points in a cluster in order to find the cluster center, k-medoids selects a point that has the smallest average distance to all the other points in the cluster.

Care needs to be taken while assigning the initial cluster. If the data is very dense with very widely separated clusters, and if the initial random centers are chosen in the same cluster, k-means may not perform very well.

Typically, k-means works if the data has star convex clusters. Refer to the following link for more information on star convex-shaped data points:

<http://mathworld.wolfram.com/StarConvex.html>

The presence of nested or other complicated clusters will result in a junk output from k-means.

The presence of outliers in the data may yield poor results. A good practice is to do a thorough data exploration in order to identify the data characteristics before running k-means.

An alternative method to initialize the centers during the beginning of the algorithm is the k-means++ method. So, instead of setting the init parameter to random, we can set it using k-means++. Refer to the following paper for k-means++:

k-means++: the advantages of careful seeding. ACM-SIAM symposium on Discrete algorithms. 2007

See also

- ▶ Working with Distance Measures recipe in Chapter 5, Data Mining - Finding a needle in a haystack

Learning vector quantization

In this recipe, we will see a model-free method for clustering the data points called Learning Vector Quantization, LVQ for short. LVQ can be used in classification tasks. Not much of an inference can be made between the target variables and prediction variables using this technique. Unlike the other methods, it is tough to make out what relationships exist between the response variable, Y, and predictor, X. They serve very well as a black box approach in many real-world scenarios.

Getting ready

LVQ is an online learning algorithm where the data points are processed one at a time. It makes a very simple intuition. Assume that we have prototype vectors identified for the different classes present in our dataset. The training points will be attracted towards the prototypes of similar classes and will repel the other prototypes.

The major steps in LVQ are as follows:

Select k initial prototype vectors for each class in the dataset. If it's a two-class problem and we decide to have two prototype vectors for each class, we will end up with four initial prototype vectors. The initial prototype vectors are selected randomly from the input dataset.

We will start our iteration. Our iteration will end when our epsilon value has reached either zero or a predefined threshold. We will decide an epsilon value and decrement the epsilon value with every iteration.

In each iteration, we will sample an input point (with replacement) and find the closest prototype vector to this point. We will use Euclidean distance to find the closest point. We will update the prototype vector of the closest point, as follows:

If the class label of the prototype vector is the same as the input data point, we will increment the prototype vector with the difference between the prototype vector and data point.

If the class label is different, we will decrement the prototype vector with the difference between the prototype vector and data point.

We will use the Iris dataset to demonstrate how LVQ works. As in some of our previous recipe, we will use the convenient data loading function from scikit-learn in order to load the Iris dataset. Iris is a well known classification dataset. However our purpose of using it here is to only demonstrate LVQ's capability. Datasets without class labels can also be used or processed by LVQ. As we are going to use Euclidean distance, we will scale the data using minmax scaling.

```
from sklearn.datasets import load_iris
import numpy as np
from sklearn.metrics import euclidean_distances

data = load_iris()
x = data['data']
y = data['target']

# Scale the variables
from sklearn.preprocessing import MinMaxScaler
minmax = MinMaxScaler()
x = minmax.fit_transform(x)
```

How to do it...

1. Let's first declare the parameters for LVQ:

```
R = 2
n_classes = 3
epsilon = 0.9
epsilon_dec_factor = 0.001
```

2. Define a class to hold the prototype vectors:

```
class prototype(object):
    """
    Class to hold prototype vectors
    """
```

```

def __init__(self, class_id, p_vector, epsilon):
    self.class_id = class_id
    self.p_vector = p_vector
    self.epsilon = epsilon

def update(self, u_vector, increment=True):
    if increment:
        # Move the prototype vector closer to input vector
        self.p_vector = self.p_vector + self.epsilon*(u_vector
- self.p_vector)
    else:
        # Move the prototype vector away from input vector
        self.p_vector = self.p_vector - self.epsilon*(u_vector
- self.p_vector)

```

3. This is the function to find the closest prototype vector for a given vector:

```

def find_closest(in_vector, proto_vectors):
    closest = None
    closest_distance = 99999
    for p_v in proto_vectors:
        distance = euclidean_distances(in_vector, p_v.p_vector)
        if distance < closest_distance:
            closest_distance = distance
            closest = p_v
    return closest

```

4. A convenient function to find the class ID of the closest prototype vector is as follows:

```

def find_class_id(test_vector, p_vectors):
    return find_closest(test_vector, p_vectors).class_id

```

5. Choose the initial K * number of classes of prototype vectors:

```

# Choose R initial prototypes for each class
p_vectors = []
for i in range(n_classes):
    # Select a class
    y_subset = np.where(y == i)
    # Select tuples for chosen class
    x_subset = x[y_subset]
    # Get R random indices between 0 and 50
    samples = np.random.randint(0, len(x_subset), R)
    # Select p_vectors
    for sample in samples:
        s = x_subset[sample]
        p = prototype(i, s, epsilon)
        p_vectors.append(p)

    print "class id \t Initial prototype\n"

```

```
for p_v in p_vectors:
    print p_v.class_id, '\t', p_v.p_vector
    print

6. Perform iteration to adjust the prototype vector in order to classify/cluster any new
incoming points using the existing data points:

while epsilon >= 0.01:
    # Sample a training instance randomly
    rnd_i = np.random.randint(0,149)
    rnd_s = x[rnd_i]
    target_y = y[rnd_i]

    # Decrement epsilon value for next iteration
    epsilon = epsilon - epsilon_dec_factor
    # Find closes prototype vector to given point
    closest_pvector = find_closest(rnd_s,p_vectors)

    # Update closes prototype vector
    if target_y == closest_pvector.class_id:
        closest_pvector.update(rnd_s)
    else:
        closest_pvector.update(rnd_s,False)
    closest_pvector.epsilon = epsilon

print "class id \t Final Prototype Vector\n"
for p_vector in p_vectors:
    print p_vector.class_id, '\t', p_vector.p_vector

7. The following is a small test to verify the correctness of our method:

predicted_y = [find_class_id(instance,p_vectors) for instance in x
]

from sklearn.metrics import classification_report

print
print classification_report(y,predicted_y,target_names=['Iris-
Setosa','Iris-Versicolour', 'Iris-Virginica'])
```

How it works...

In step 1, we initialize the parameters for the algorithm. We have chosen our R value as two, that is, we have two prototype vectors per class label. The Iris dataset is a three-class problem, so we have six prototype vectors in total. We must choose our epsilon value and epsilon decrement factor.

We then define a data structure to hold the details of our prototype vector in step 2. Our class stores the following for each point in the dataset:

```
self.class_id = class_id  
self.p_vector = p_vector  
self.epsilon = epsilon
```

The class id to which the prototype vector belongs is the vector itself and the epsilon value. It also has a function update that is used to change the prototype values:

```
def update(self,u_vector,increment=True) :  
    if increment:  
        # Move the prototype vector closer to input vector  
        self.p_vector = self.p_vector + self.epsilon*(u_vector - self.p_  
vector)  
    else:  
        # Move the prototype vector away from input vector  
        self.p_vector = self.p_vector - self.epsilon*(u_vector - self.p_  
vector)
```

In step 3, we define the following function, which takes any given vector as the input and a list of all the prototype vectors. Out of all the prototype vectors, this function returns the closest prototype vector to the given vector:

```
for p_v in proto_vectors:  
    distance = euclidean_distances(in_vector,p_v.p_vector)  
    if distance < closest_distance:  
        closest_distance = distance  
        closest = p_v
```

As you can see, it loops through all the prototype vectors to find the closest one. It uses Euclidean distance to measure the similarity.

Step 4 is a small function that can return the class ID of the closest prototype vector to the given vector.

Now that we have finished all the required preprocessing for the LVQ algorithm, we can move on to the actual algorithm in step 5. For each class, we must select the initial prototype vectors. We then select R random points from each class. The outer loop goes through each class, and for each class, we select R random samples and create our prototype object, as follows:

```
samples = np.random.randint(0,len(x_subset),R)  
# Select p_vectors  
for sample in samples:  
    s = x_subset[sample]  
    p = prototype(i,s,epsilon)  
    p_vectors.append(p)
```

In step 6, we increment or decrement the prototype vectors iteratively. We loop continuously till our epsilon value falls below a threshold of 0.01.

We then randomly sample a point from our dataset, as follows:

```
# Sample a training instance randomly
rnd_i = np.random.randint(0,149)
rnd_s = x[rnd_i]
target_y = y[rnd_i]
```

The point and its corresponding class ID have been retrieved.

We can then find the closest prototype vector to this point, as follows:

```
closest_pvector = find_closest(rnd_s,p_vectors)
```

If the current point's class ID matches the prototype's class ID, we call the `update` method, with the increment set to True, or else we will call the `update` with the increment set to False:

```
# Update closes prototype vector
if target_y == closest_pvector.class_id:
    closest_pvector.update(rnd_s)
else:
    closest_pvector.update(rnd_s,False)
```

Finally, we update the epsilon value for the closest prototype vector:

```
closest_pvector.epsilon = epsilon
```

We can print the prototype vectors in order to look at them manually:

```
print "class id \t Final Prototype Vector\n"
for p_vector in p_vectors:
    print p_vector.class_id, '\t', p_vector.p_vector
```

In step 7, we put our prototype vectors into action to do some predictions:

```
predicted_y = [find_class_id(instance,p_vectors) for instance in x ]
```

We can get the predicted class ID using the `find_class_id` function. We pass a point and all the learned prototype vectors to it to get the class ID.

Finally, we give our predicted output in order to generate a classification report:

```
print classification_report(y,predicted_y,target_names=['Iris-Setosa','Iris-Versicolour', 'Iris-Virginica'])
```

The classification report function is a convenient function provided by the scikit-learn library to view the classification accuracy scores:

	precision	recall	f1-score	support
Iris-Setosa	1.00	1.00	1.00	50
Iris-Versicolour	0.92	0.98	0.95	50
Iris-Virginica	0.98	0.92	0.95	50
avg / total	0.97	0.97	0.97	150

You can see that we have done pretty well with our classification. Keep in mind that we did not keep a separate test set. Never measure the accuracy of your model based on the training data. Always use a test set that is unseen by the training routines. We did it only for illustration purposes.

There's more...

Keep in mind that this technique does not involve any optimization criteria as in the other classification methods. Hence, it is very difficult to judge how good the prototype vectors have been generated.

In our recipe, we initialized the prototype vectors as random values. You can use the k-means algorithm to initialize the prototype vectors.

See also

- ▶ Clustering of data using K-Means recipe in *Chapter 5, Data Mining - Finding a needle in a haystack*

Finding outliers in univariate data

Outliers are data points that are far away from the other data points in your data. They have to be handled carefully in data science applications. Including them in some of your algorithms unknowingly may lead to wrong results or conclusions. It is very important to account for them properly and have the right algorithms in order to handle them.

"Outlier detection is an extremely important problem with a direct application in a wide variety of application domains, including fraud detection (Bolton, 2002), identifying computer network intrusions and bottlenecks (Lane, 1999), criminal activities in e-commerce and detecting suspicious activities (Chiu, 2003)."

- Jayakumar and Thomas, A New Procedure of Clustering Based on Multivariate Outlier Detection (*Journal of Data Science* 11(2013), 69-84)

We will look at the detection of outliers in univariate data in this recipe and then move on to look at outliers in multivariate and text data.

Getting ready

In this recipe, we will look at the following three methods for outlier detection in univariate data:

- ▶ Median absolute deviation
- ▶ Mean plus or minus three standard deviation

Let's see how we can leverage these methods to spot outliers in univariate data. Before we jump into the next section, let's create a dataset with outliers so that we can evaluate our method empirically:

```
import numpy as np
import matplotlib.pyplot as plt

n_samples = 100
fraction_of_outliers = 0.1
number_inliers = int( (1-fraction_of_outliers) * n_samples )
number_outliers = n_samples - number_inliers
```

We will create 100 data points, and 10 percent of them will be outliers:

```
# Get some samples from a normal distribution
normal_data = np.random.randn(number_inliers,1)
```

We will use the `randn` function in the `random` module of NumPy to generate our inliers. This will be a sample from a distribution with a mean of zero and a standard deviation of one. Let's verify the mean and standard deviation of our sample:

```
# Print the mean and standard deviation
# to confirm the normality of our input data.
mean = np.mean(normal_data, axis=0)
std = np.std(normal_data, axis=0)
print "Mean = (%.2f) and Standard Deviation (%.2f)" % (mean[0], std[0])
```

We will calculate the mean and standard deviation with the functions from NumPy and print the output. Let's inspect the output:

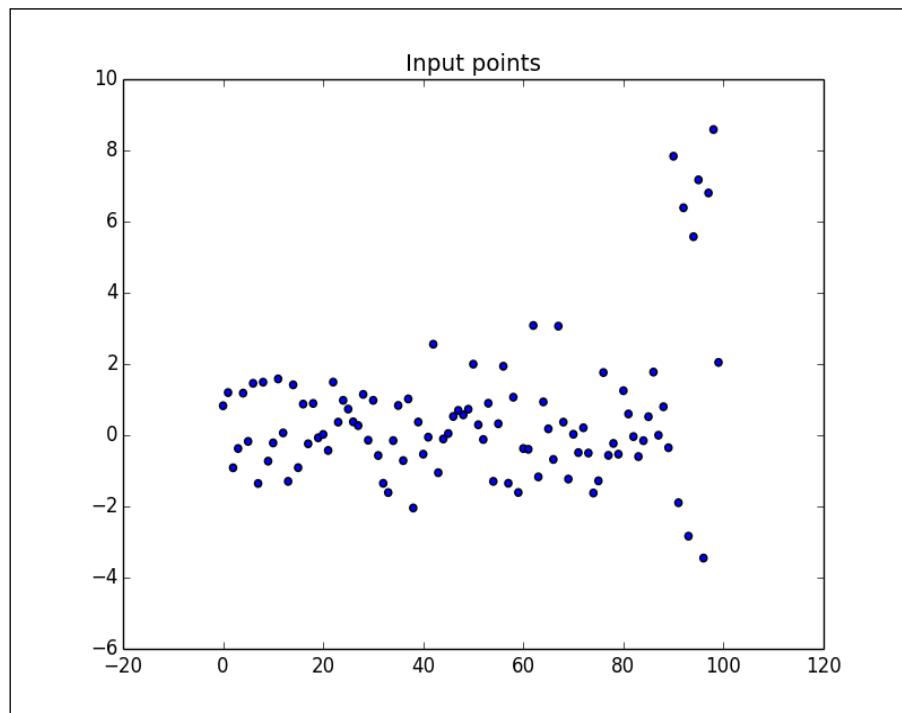
```
Mean = (0.24) and Standard Deviation (0.90)
```

As you can see, the mean is close to zero and the standard deviation is close to one.

Now, let's create the outliers. This will be 10 percent of the whole dataset, that is, 10 points, given that our sample size is 100. As you can see, we sampled our outliers from a uniform distribution between -9 and 9. Any points between this range will have an equal chance of being selected. We will concatenate our inlier and outlier data. It will be good to see the data with a scatter plot before we run our outlier detection program:

```
# Create outlier data
outlier_data = np.random.uniform(low=-9,high=9,size=(number_
outliers,1))
total_data = np.r_[normal_data,outlier_data]
print "Size of input data = (%d,%d) "%(total_data.shape)
# Eyeball the data
plt.cla()
plt.figure(1)
plt.title("Input points")
plt.scatter(range(len(total_data)),total_data,c='b')
```

Let's look at the graph that is generated:



Our y axis is the actual values that we generated and our x axis is a running count. It will be a good exercise to mark the points that you feel are outliers. We can later compare our program output with your manual selections.

How to do it...

1. Let's start with the median absolute deviation. Then we will plot our values, with the outliers marked in red:

```
# Median Absolute Deviation
median = np.median(total_data)
b = 1.4826
mad = b * np.median(np.abs(total_data - median))
outliers = []
# Useful while plotting
outlier_index = []
print "Median absolute Deviation = %.2f"%(mad)
lower_limit = median - (3*mad)
upper_limit = median + (3*mad)
print "Lower limit = %0.2f, Upper limit = %0.2f"%(lower_limit,upper_limit)
for i in range(len(total_data)):
    if total_data[i] > upper_limit or total_data[i] < lower_limit:
        print "Outlier %0.2f"%(total_data[i])
        outliers.append(total_data[i])
        outlier_index.append(i)

plt.figure(2)
plt.title("Outliers using mad")
plt.scatter(range(len(total_data)),total_data,c='b')
plt.scatter(outlier_index,outliers,c='r')
plt.show()
```

2. Moving on to the mean plus or minus three standard deviation, we will plot our values, with the outliers colored in red:

```
# Standard deviation
std = np.std(total_data)
mean = np.mean(total_data)
b = 3
outliers = []
outlier_index = []
lower_lmt = mean-b*std
upper_lmt = mean+b*std
print "Lower limit = %0.2f, Upper limit = %0.2f"%(lower_lmt,upper_lmt)
for i in range(len(total_data)):
    x = total_data[i]
    if x > upper_lmt or x < lower_lmt:
        print "Outlier %0.2f"%(total_data[i])
```

```
outliers.append(total_data[i])
outlier_index.append(i)

plt.figure(3)
plt.title("Outliers using std")
plt.scatter(range(len(total_data)),total_data,c='b')
plt.scatter(outlier_index,outliers,c='r')
plt.savefig("B04041 04 10.png")
plt.show()
```

How it works...

In step 1, we use the median absolute deviation to detect the outliers in the data:

```
median = np.median(total_data)
b = 1.4826
mad = b * np.median(np.abs(total_data - median))
```

We first calculate the median value of our dataset using the median function from NumPy. Next, we declare a variable with a value of 1.4826. This is a constant to be multiplied with the absolute deviation from the median. Finally, we calculate the median of absolute deviations of each entry from the median value and multiply it with the constant, b.

Any point that is more than or less than three times the median absolute deviation is deemed as an outlier for our method:

```
lower_limit = median - (3*mad)
upper_limit = median + (3*mad)

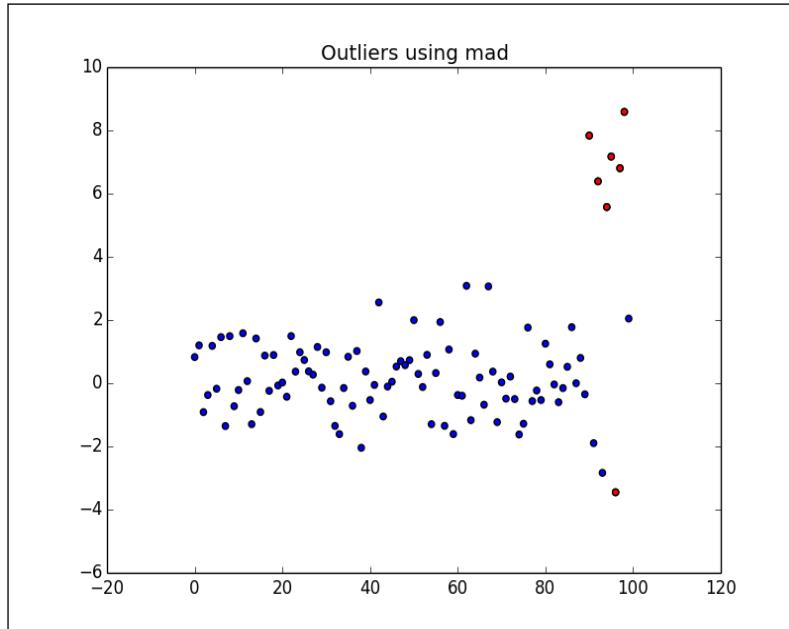
print "Lower limit = %0.2f, Upper limit = %0.2f"%(lower_limit,upper_
limit)
```

We then calculate the lower and upper limits of the median absolute deviation, as shown previously, and classify every point as either an outlier or inlier, as follows:

```
for i in range(len(total_data)):
    if total_data[i] > upper_limit or total_data[i] < lower_limit:
        print "Outlier %0.2f"%(total_data[i])
        outliers.append(total_data[i])
        outlier_index.append(i)
```

Finally, we have all our outlier points stored in a list by the name of outliers. We must also store the index of the outliers in a separate list called outlier_index. This is done for the ease of plotting, as you will see in the next step.

We then plot the original points and outliers. The plot looks as follows:



The points marked in red are classified as outliers by the algorithm.

In step 3, we code up the second algorithm, mean plus or minus three standard deviation:

```
std = np.std(total_data)
mean = np.mean(total_data)
b = 3
```

We then calculate the standard deviation and mean of our dataset. Here, you can see that we have set our $b = 3$. As the name of our algorithm suggests, we will need a standard deviation of three, and this b is used for the same:

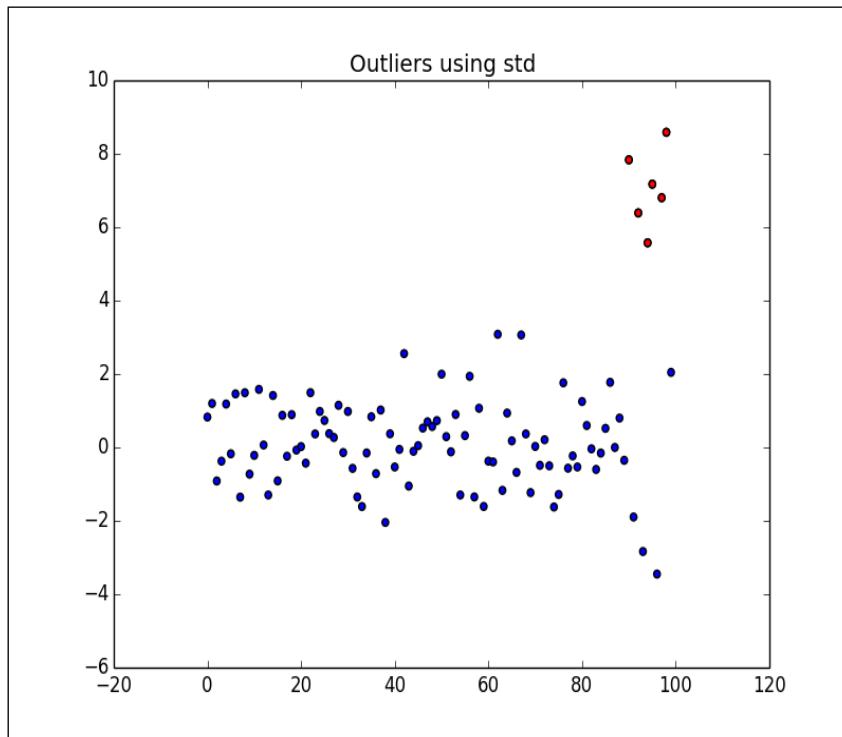
```
lower_limit = mean-b*std
upper_limit = mean+b*std

print "Lower limit = %0.2f, Upper limit = %0.2f"%(lower_limit,upper_
limit)

for i in range(len(total_data)):
    x = total_data[i]
    if x > upper_limit or x < lower_limit:
        print "Outlier %0.2f"%(total_data[i])
        outliers.append(total_data[i])
        outlier_index.append(i)
```

We can calculate the lower and upper limits as the mean minus three times the standard deviation. Using these values, we can then classify every point as either an outlier or inlier in the for loop. We then add all the outliers and their indices to the two lists, outliers and outlier_index, to plot.

Finally, we plot the outliers:



There's more...

As per the definition of outliers, outliers in a given dataset are those points that are far away from the other points in the data source. The estimates of the center of the dataset and the spread of the dataset can be used to detect the outliers. In the methods that we outlined in this recipe, we used the mean and median as the estimates for the center of the data and standard deviation, and the median absolute deviation as the estimates for the spread. Spread is also called scale.

Let's do a little bit of rationalization about why our methods work in the detection of the outliers. Let's start with the method of using standard deviation. For Gaussian data, we know that 68.27 percent of the data lies within one standard deviation, 95.45 percent in two, and 99.73 percent lies in three. Thus, according to our rule that any point that is more than three standard deviations from the mean is classified as an outlier. However, this method is not robust. Let's look at a small example.

Let's sample eight data points from a normal distribution, with the mean as zero and the standard deviation as one.

Let's use the convenient function from NumPy `.random` to generate our numbers:

```
np.random.randn(8)
```

This gives us the following numbers:

```
-1.76334861, -0.75817064, 0.44468944, -0.07724717,  
0.12951944, 0.43096092, -0.05436724, -0.23719402
```

Let's add two outliers to it manually, for example, 45 and 69, to this list.

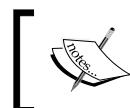
Our dataset now looks as follows:

```
-1.763348607322289, -0.7581706357821458, 0.4446894368956213,  
-0.07724717210195432, 0.1295194428816003, 0.4309609200681169,  
-0.05436724238743103, -0.23719402072058543, 45, 69
```

The mean of the preceding dataset is 11.211 and the standard deviation is 23.523.

Let's look at the upper rule, $\text{mean} + 3 * \text{std}$. This is $11.211 + 3 * 23.523 = 81.78$.

Now, according to this upper bound rule, both the points, 45 and 69, are not outliers! Both the mean and the standard deviation are non-robust estimators of the center and scale of the dataset, as they are extremely sensitive to outliers. If we replace one of the points with an extreme point in a dataset with n observations, it will completely change the estimate of the mean and the standard deviation. This property of the estimators is called the finite sample breakdown point.



The finite sample breakdown point is defined as the proportion of the observations in a sample that can be replaced before the estimator fails to describe the data accurately.

Thus, for the mean and standard deviation, the finite sample breakdown point is 0 percent because in a large sample, replacing even a single point would change the estimators drastically.

In contrast, the median is a more robust estimate. The median is the middle observation in a finite set of observations that is sorted in an ascending order. For the median to change drastically, we have to replace half of the observations in the data that are far away from the median. This gives you a 50 percent finite sample breakdown point for the median.

The median absolute deviation method is attributed to the following paper:

Leys, C., et al., *Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median*, *Journal of Experimental Social Psychology* (2013), <http://dx.doi.org/10.1016/j.jesp.2013.03.013>.

See also

- ▶ *Performing summary statistics and plots* recipe in Chapter 1, *Using Python for Data Science*

Discovering outliers using the local outlier factor method

The Local Outlier Factor (LOF) is an outlier detection algorithm that detects the outliers based on comparing the local density of the data instance with its neighbors. It does this in order to decide if the data instance belongs to a region of similar density. It can detect an outlier in a dataset, in such circumstances where the number of clusters are unknown and the clusters are of different density and sizes. It's inspired by the KNN (K-Nearest Neighbors) algorithm and is widely used.

Getting ready

In the previous recipe, we looked at univariate data. In this one, we will use multivariate data and try to find outliers. Let's use a very small dataset to understand the LOF algorithm for outlier detection.

We will create a 5 X 2 matrix, and looking at the data, we know that the last tuple is an outlier. Let's also plot it as a scatter plot:

```
from collections import defaultdict
import numpy as np

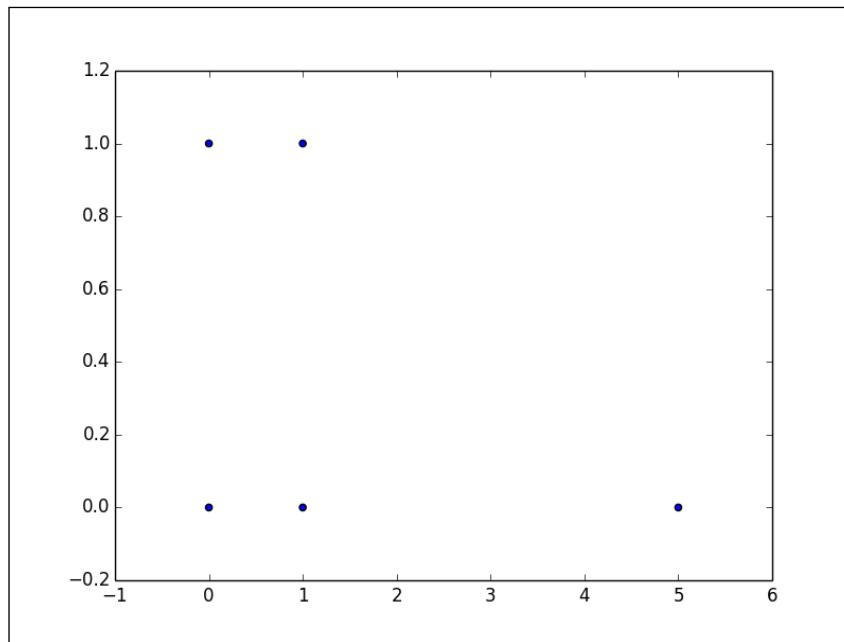
instances = np.matrix([[0,0],[0,1],[1,1],[1,0],[5,0]])

import numpy as np
import matplotlib.pyplot as plt

x = np.squeeze(np.asarray(instances[:,0]))
y = np.squeeze(np.asarray(instances[:,1]))
```

```
plt.cla()
plt.figure(1)
plt.scatter(x,y)
plt.show()
```

The plot looks as follows:



LOF works by calculating the local density of each point. Based on the distance of k-nearest neighbors of a point, the local density of the point is estimated. By comparing the local density of the point with the densities of its neighbors, outliers are detected. Outliers have a low density compared with their neighbors.

We will need to go through some term definitions in order to understand LOF:

- ▶ The k-distance of object P is the distance between the object P and its kth nearest neighbor. K is a parameter of the algorithm.
- ▶ The k-distance neighborhood of P is the list of all the objects, Q, whose distance from P is either less than or equal to the distance between P and its kth nearest object.
- ▶ The reachability distance from P to Q is defined as the maximum of the distance between P and its kth nearest neighbor, and the distance between P and Q. The following notation may help clarify this:

Reachability distance ($P \rightarrow Q$) = $\max(K\text{-Distance}(P), Distance(P, Q))$

- ▶ The Local Reachability Density of P (LRD(P)) is the ratio of the k-distance neighborhood of P and the sum of the reachability distance of k and its neighborhood.
- ▶ The Local Outlier Factor of P (LOF(P)) is the average of the ratio of the local reachability of P and those of P's k-nearest neighbors.

How to do it...

1. Let's get the pairwise distance between the points:

```
k = 2
distance = 'manhattan'
```

```
from sklearn.metrics import pairwise_distances
dist = pairwise_distances(instances, metric=distance)
```

2. Let's calculate the k-distance. We will use heapq and get the k-nearest neighbors:

```
# Calculate K distance
import heapq
k_distance = defaultdict(tuple)
# For each data point
for i in range(instances.shape[0]):
    # Get its distance to all the other points.
    # Convert array into list for convenience
    distances = dist[i].tolist()
    # Get the K nearest neighbours
    ksmallest = heapq.nsmallest(k+1, distances)[1:] [k-1]
    # Get their indices
    ksmallest_idx = distances.index(ksmallest)
    # For each data point store the K th nearest neighbour and its
    # distance
    k_distance[i] = (ksmallest, ksmallest_idx)
```

3. Calculate the k-distance neighborhood:

```
def all_indices(value, inlist):
    out_indices = []
    idx = -1
    while True:
        try:
            idx = inlist.index(value, idx+1)
            out_indices.append(idx)
        except ValueError:
            break
    return out_indices
# Calculate K distance neighbourhood
import heapq
```

```
k_distance_neig = defaultdict(list)
# For each data point
for i in range(instances.shape[0]):
    # Get the points distances to its neighbours
    distances = dist[i].tolist()
    print "k distance neighbourhood",i
    print distances
    # Get the 1 to K nearest neighbours
    ksmallest = heapq.nsmallest(k+1,distances) [1:]
    print ksmallest
    ksmallest_set = set(ksmallest)
    print ksmallest_set
    ksmallest_idx = []
    # Get the indices of the K smallest elements
    for x in ksmallest_set:
        ksmallest_idx.append(all_indices(x,distances))
    # Change a list of list to list
    ksmallest_idx = [item for sublist in ksmallest_idx for item in
    sublist]
    # For each data point store the K distance neighbourhood
    k_distance_neig[i].extend(zip(ksmallest,ksmallest_idx))
```

4. Then, calculate the reachability distance and LRD:

```
#Local reachable density
local_reach_density = defaultdict(float)
for i in range(instances.shape[0]):
    # LRDs numerator, number of K distance neighbourhood
    no_neighbours = len(k_distance_neig[i])
    denom_sum = 0
    # Reachability distance sum
    for neigh in k_distance_neig[i]:
        # maximum(K-Distance(P), Distance(P,Q))
        denom_sum+=max(k_distance[neigh[1]][0],neigh[0])
    local_reach_density[i] = no_neighbours/(1.0*denom_sum)
```

5. Calculate LOF:

```
lof_list = []
#Local Outlier Factor
for i in range(instances.shape[0]):
    lrd_sum = 0
    rdist_sum = 0
    for neigh in k_distance_neig[i]:
        lrd_sum+=local_reach_density[neigh[1]]
        rdist_sum+=max(k_distance[neigh[1]][0],neigh[0])
    lof_list.append((i,lrd_sum*rdist_sum))
```

How it works...

In step 1, we select our distance metric to be Manhattan and our k value as two. We are looking at the second nearest neighbor for our data point.

We must then proceed to calculate the pairwise distance between our tuples. The pairwise similarity is stored in the dist matrix. As you can see, the shape of dist is as follows:

```
>>> dist.shape  
(5, 5)  
>>>
```

It is a 5 X 5 matrix, where the rows and columns are individual tuples and the cell value indicates the distance between them.

In step 2, we then import heapq:

```
import heapq
```

heapq is a data structure that is also known as a priority queue. It is similar to a regular queue except that each element is associated with a priority, and an element with a high priority is served before an element with a low priority.

Refer to the Wikipedia link for more information on priority queues:

http://en.wikipedia.org/wiki/Priority_queue.

The Python heapq documentation can be found at <https://docs.python.org/2/library/heappq.html>.

```
k_distance = defaultdict(tuple)
```

Next, we define a dictionary where the key is the tuple ID and the value is the distance of the tuple to its kth nearest neighbor. In our case, it should be the second nearest neighbor.

We then enter a for loop in order to find the kth nearest neighbor's distance for each of the data points:

```
distances = dist[i].tolist()
```

From our distance matrix, we extract the ith row. As you can see, the ith row captures the distance between the object i and all the other objects. Remember that the cell value (i,i) holds the distance to itself. We need to ignore this in the next step. We must convert the array to a list for our convenience. Let's try to understand this with an example. The distance matrix looks as follows:

```
>>> dist  
array([[ 0.,  1.,  2.,  1.,  5.],  
       [ 1.,  0.,  1.,  2.,  6.],
```

```
[ 2.,  1.,  0.,  1.,  5.],  
[ 1.,  2.,  1.,  0.,  4.],  
[ 5.,  6.,  5.,  4.,  0.]])
```

Let's assume that we are in the first iteration of our for loop and hence, our $i = 0$. (remember that the Python indexing starts with 0).

So, now our distances list will look as follows:

```
[ 0.,  1.,  2.,  1.,  5.]
```

From this, we need the k th nearest neighbor, that is, the second nearest neighbor, as we have set $K = 2$ at the beginning of the program.

Looking at it, we can see that both index 1 and index 3 can be our the k th nearest neighbor as both have a value of 1.

Now, we use the `heapq.nsmallest` function. Remember that we had mentioned that `heapq` is a normal queue but with a priority associated with each element. The value of the element is the priority in this case. When we say that give me the n smallest, `heapq` will return the smallest elements:

```
# Get the Kth nearest neighbours  
ksmallest = heapq.nsmallest(k+1,distances) [1:] [k-1]
```

Let's look at what the `heapq.nsmallest` function does:

```
>>> help(heapq.nsmallest)  
Help on function nsmallest in module heapq:  
  
nsmallest(n, iterable, key=None)  
    Find the n smallest elements in a dataset.  
  
    Equivalent to: sorted(iterable, key=key) [:n]
```

It returns the n smallest elements from the given dataset. In our case, we need the second nearest neighbor. Additionally, we need to avoid (i,i) as mentioned previously. So we must pass $n = 3$ to `heapq.nsmallest`. This ensures that it returns the three smallest elements. We then subset the list to exclude the first element (see `[1:]` after `nsmallest` function call) and finally retrieve the second nearest neighbor (see `[k-1]` after `[1:]`).

We must also get the index of the second nearest neighbor of i and store it in our dictionary:

```
# Get their indices  
ksmallest_idx = distances.index(ksmallest)  
# For each data point store the K th nearest neighbour and its  
# distance  
k_distance[i]=(ksmallest, ksmallest_idx)
```

Let's print our dictionary:

```
print k_distance
defaultdict(<type 'tuple'>, {0: (1.0, 1), 1: (1.0, 0), 2: (1.0, 1), 3:
(1.0, 0), 4: (5.0, 0)})
```

Our tuples have two elements: the distance, and the index of the elements in the distances array. So, for instance 0, the second nearest neighbor is the element in index 1.

Having calculated the k-distance for all our data points, we then move on to find the k-distance neighborhood.

In step 3, we find the k-distance neighborhood for each of our data points:

```
# Calculate K distance neighbourhood
import heapq
k_distance_neig = defaultdict(list)
```

Similar to our previous step, we import the heapq module and declare a dictionary that is going to hold our k-distance neighborhood details. Let's recap what the k-distance neighborhood is:

The k-distance neighborhood of P is the list of all the objects, Q, whose distance from P is either less than or equal to the distance between P and its kth nearest object:

```
distances = dist[i].tolist()
# Get the 1 to K nearest neighbours
ksmallest = heapq.nsmallest(k+1,distances)[1:]
ksmallest_set = set(ksmallest)
```

The first two lines should be familiar to you. We did this in our previous step. Look at the second line. Here, we invoked n smallest again with n=3 in our case (K+1), but we selected all the elements in the output list except the first one. (Guess why? The answer is in the previous step.)

Let's see it in action by printing the values. As usual, in the loop, we assume that we are seeing the first data point or tuple where i=0.

Our distances list is as follows:

```
[0.0, 1.0, 2.0, 1.0, 5.0]
```

Our heapq.nsmallest function returns the following:

```
[1.0, 1.0]
```

These are 1 to k-nearest neighbor's distances. We need to find their indices, a simple list. `index` function will only return the first match, so we will write the `all_indices` function in order to retrieve all the indices:

```
def all_indices(value, inlist):
    out_indices = []
    idx = -1
    while True:
        try:
            idx = inlist.index(value, idx+1)
            out_indices.append(idx)
        except ValueError:
            break
    return out_indices
```

With a value and list, `all_indices` will return all the indices where the value occurs in the list. We must convert our k smallest to a set:

```
ksmallest_set = set(ksmallest)
```

So, [1.0,1.0] becomes a set ([1.0]). Now, using a for loop, we can find all the indices of the elements:

```
# Get the indices of the K smallest elements
for x in ksmallest_set:
    ksmallest_idx.append(all_indices(x,distances))
```

We get two indices for 1.0; they are 1 and 2:

```
ksmallest_idx = [item for sublist in ksmallest_idx for item in
sublist]
```

The next for loop is to convert a list of the lists to a list. The `all_indices` function returns a list, and we then append this list to the `ksmallest_idx` list. Hence, we flatten it using the next for loop.

Finally, we add the k smallest neighborhood to our dictionary:

```
k_distance_neig[i].extend(zip(ksmallest, ksmallest_idx))
```

We then add tuples where the first item in the tuple is the distance and the second item is the index of the nearest neighbor. Let's print the k-distance neighborhood dictionary:

```
defaultdict(<type 'list'>, {0: [(1.0, 1), (1.0, 3)], 1: [(1.0, 0),
(1.0, 2)], 2: [(1.0, 1), (1.0, 3)], 3: [(1.0, 0), (1.0, 2)], 4: [(4.0,
3), (5.0, 0)]})
```

In step 4, we calculate the LRD. The LRD is calculated using the reachability distance. Let's recap both the definitions:

- ▶ The reachability distance from P to Q is defined as the maximum of the distance between P and its kth nearest neighbor, and the distance between P and Q. The following notation may help clarify this:

```
Reachability distance (P & Q) = > maximum(K-Distance(P),  
Distance(P,Q))
```

- ▶ The Local Reachability density of P (LRD(P)) is the ratio of the k-distance neighborhood of P and the sum of the reachability distance of k and its neighborhood:

```
#Local reachable density  
local_reach_density = defaultdict(float)
```

We will first declare a dictionary in order to store the LRD:

```
for i in range(instances.shape[0]):  
    # LRDs numerator, number of K distance neighbourhood  
    no_neighbours = len(k_distance_neig[i])  
    denom_sum = 0  
    # Reachability distance sum  
    for neigh in k_distance_neig[i]:  
        # maximum(K-Distance(P), Distance(P,Q))  
        denom_sum+=max(k_distance[neigh[1]][0],neigh[0])  
    local_reach_density[i] = no_neighbours/(1.0*denom_sum)
```

For every point, we will first find the k-distance neighborhood of that point. For example, for $i = 0$, the numerator would be $\text{len}(k_distance_neig[0])$, 2.

Now, in the inner for loop, we calculate the denominator. We then calculate the reachability distance for each k-distance neighborhood point. The ratio is stored in the `local_reach_density` dictionary.

Finally, in step 5, we calculate the LOF for each point:

```
for i in range(instances.shape[0]):  
    lrd_sum = 0  
    rdist_sum = 0  
    for neigh in k_distance_neig[i]:  
        lrd_sum+=local_reach_density[neigh[1]]  
        rdist_sum+=max(k_distance[neigh[1]][0],neigh[0])  
    lof_list.append((i,lrd_sum*rdist_sum))
```

For each data point, we calculate the LRD sum of its neighbor and the reachability distance sum with its neighbor, and multiply them to get the LOF.

The point with a very high LOF is considered an outlier. Let's print `lof_list`:

```
[(0, 4.0), (1, 4.0), (2, 4.0), (3, 4.0), (4, 18.0)]
```

As you can see, the last point has a very high LOF compared with the others and hence, it's an outlier.

There's more...

You can refer to the following paper in order to understand more about LOF:

LOF: Identifying Density-Based Local Outliers

Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, Jörg Sander

Proc. ACM SIGMOD 2000 Int. Conf. On Management of Data, Dallas, TX, 2000

6

Machine Learning 1

In this chapter, we will cover the following topics:

- ▶ Preparing data for model building
- ▶ Finding the nearest neighbors
- ▶ Classifying documents using Naïve Bayes
- ▶ Building decision trees to solve multiclass problems

Introduction

In this chapter, we will look at supervised learning techniques. In the previous chapter, we saw unsupervised techniques including clustering and learning vector quantization. We will start with a classification problem and then proceed to regression. The input for a classification problem is a set of records or instances in the next chapter.

Each record or instance can be written as a set (X, y) , where X is a set of attributes and y is a corresponding class label.

Learning a target function, F , that maps each record's attribute set to one of the predefined class label, y , is the job of a classification algorithm.

The general steps for a classification algorithm are as follows:

1. Find an appropriate algorithm
2. Learn a model using a training set, and validate the model using a test set
3. Apply the model to predict any unseen instance or record

The first step is to identify the right classification algorithm. There is no prescribed way of choosing the right algorithm, it comes from repeated trial and error. After choosing the algorithm, a training and a test set is created, which is provided to the algorithm to learn a model, that is, a target function F , as defined previously. After creating the model using a training set, a test set is used to validate the model. Usually, we use a confusion matrix to validate the model. We will discuss more about confusion matrices in our recipe: Finding the nearest neighbors.

We will begin with a recipe that will show us how to divide our input dataset into training and test sets. We will follow this with a lazy learner algorithm for classification, called K-Nearest Neighbor. We will then look at Naïve Bayes classifiers. We will venture into a recipe that deals with multiclass problems using decision trees. Our choice of algorithms in this chapter is not random. All the three algorithms that we will cover in this chapter are capable of handling multiclass problems, in addition to binary problems. In multiclass problems, we have more than two class labels to which the instances can belong.

Preparing data for model building

In this recipe, we will look at how to create a train and a test dataset from the given dataset for the classification problem. A test dataset is never shown to the model. In real-world scenarios, we typically build another dataset called dev. Dev stands for development dataset: a dataset that we can use to continuously tune our model during successive runs. The model is trained using the train set, and model performance metrics such as accuracy are measured in dev. Based on this result, the model is further tuned in case improvements are required. In later chapters, we will cover recipes that can do more sophisticated data splitting than just a simple train test split.

Getting ready

We will use the Iris dataset for this recipe. It's easy to demonstrate the concept with this dataset as we are familiar with it because we have used it for many of our previous recipes.

How to do it...

```
# Load the necessary Library
from sklearn.cross_validation import train_test_split
from sklearn.datasets import load_iris
import numpy as np

def get_iris_data():
    """
    Returns Iris dataset
    """

```

```
# Load iris dataset
data = load_iris()

# Extract the dependend and independent variables
# y is our class label
# x is our instances/records
x     = data['data']
y     = data['target']

# For ease we merge them
# column merge
input_dataset = np.column_stack([x,y])

# Let us shuffle the dataset
# We want records distributed randomly
# between our test and train set

np.random.shuffle(input_dataset)

return input_dataset

# We need 80/20 split.
# 80% of our records for Training
# 20% Remaining for our Test set
train_size = 0.8
test_size  = 1-train_size

# get the data
input_dataset = get_iris_data()
# Split the data
train,test = train_test_split(input_dataset,test_size=test_size)

# Print the size of original dataset
print "Dataset size ",input_dataset.shape
# Print the train/test split
print "Train size ",train.shape
print "Test size",test.shape
```

This was pretty simple. Let's see if the class labels are proportionately distributed between the training and the test sets. This is a typical class imbalance problem:

```
def get_class_distribution(y):

    """
    Given an array of class labels
    Return the class distribution
    """

```

```
distribution = {}
set_y = set(y)
for y_label in set_y:
    no_elements = len(np.where(y == y_label)[0])
    distribution[y_label] = no_elements
dist_percentage = {class_label: count/(1.0*sum(distribution.
values())) for class_label, count in distribution.items()}
return dist_percentage

def print_class_label_split(train,test):
"""
Print the class distribution
in test and train dataset
"""
y_train = train[:, -1]

train_distribution = get_class_distribution(y_train)
print "\nTrain data set class label distribution"
print "=====\\n"
for k,v in train_distribution.items():
    print "Class label =%d, percentage records =%.2f"%(k,v)

y_test = test[:, -1]

test_distribution = get_class_distribution(y_test)

print "\nTest data set class label distribution"
print "=====\\n"

for k,v in test_distribution.items():
    print "Class label =%d, percentage records =%.2f"%(k,v)

print_class_label_split(train,test)
```

Let's see how we distribute the class labels uniformly between the train and the test sets:

```
# Perform Split the data
stratified_split = StratifiedShuffleSplit(input_dataset[:, -1], test_
size=test_size, n_iter=1)

for train_idx, test_idx in stratified_split:
    train = input_dataset[train_idx]
    test = input_dataset[test_idx]
    print_class_label_split(train, test)
```

How it works...

After we import the necessary library modules, we must write a convenient function, `get_iris_data()`, which will return the Iris dataset. We then column concatenate the `x` and `y` arrays into a single array called `input_dataset`. We then shuffle the dataset so that the records can be distributed randomly to the test and the train datasets. The function returns a single array of both the instances and the class labels.

We want to include 80 percent of the record in our training dataset, and use the remaining as our test dataset. The `train_size` and `test_size` variables hold a percentage of the values, which should be in the training and testing dataset.

We must call the `get_iris_data()` function in order to get the input data. We then leverage the `train_test_split` function from scikit-learn's `cross_validation` model to split the input dataset into two.

Finally, we can print the size of the original dataset, followed by the test and the train datasets:

```
Compare Data Set Size
=====
Original Dataset size  (150, 5)
Train size  (120, 5)
Test  size (30, 5)
```

Our original dataset has 150 rows and five columns. Remember that there are only four attributes; the fifth column is the class label. We had column concatenated `x` and `y`.

As you can see, 80 percent of the 150 rows, that is, 120 records, have been assigned to our training set. We have shown how we can easily split our input data into the train and the test sets.

Remember this is a classification problem. The algorithm should be trained to predict the correct class label for a given unknown instance or record. For this, we need to provide the algorithm and an equal distribution of all the classes during training. The Iris dataset is a three-class problem. We should have equal representation from all the three classes. Let's see if our method has taken care of this.

We must define a function called `get_class_distribution`, which takes a single `y` parameter's array of class labels. This function returns a dictionary, where the key is the class label and the value is a percentage of the number of records for this distribution. Thus, this dictionary gives us the distribution of the class labels. We must call this function in the following function to get to know what our class distribution is in the train and the test datasets.

The `print_class_label_split` function is self-explanatory. We must pass the train and the test datasets as the argument. As we have concatenated our `x` and `y`, the last column is our class label. We then extract the train and test class labels in `y_train` and `y_test`. We pass them to `get_class_distribution` to get a dictionary of the class labels and their distribution, and finally, we print it.

We can then finally invoke `print_class_label_split`, and our output should look as follows:

```
Train data set class label distribution
=====
Class label =0, percentage records =0.36
Class label =1, percentage records =0.32
Class label =2, percentage records =0.33

Test data set class label distribution
=====
Class label =0, percentage records =0.23
Class label =1, percentage records =0.40
Class label =2, percentage records =0.37
```

Let's now examine the output. As you can see, our training set has a different distribution of the class labels compared with the test set. Exactly 40 percent of the instances in the test set belong to class label 1. This is not the right way to do the split. We should have an equal distribution in both the training and the test datasets.

In the final piece of code, we leverage `StratifiedShuffleSplit` from scikit-learn in order to achieve equal class distribution in the training and the test sets. Let's examine the parameters of `StratifiedShuffleSplit`:

```
stratified_split = StratifiedShuffleSplit(input_dataset[:, -1], test_size=test_size, n_iter=1)
```

The first parameter is the input dataset. We pass all the rows and the last column. Our test size is defined by the `test_size` variable, which we had initially declared. We can assume that we need only one split using the `n_iter` variable. We then proceed to invoke `print_class_label_split` to print the class label distribution. Let's examine the output:

```
Train data set class label distribution
=====
Class label =0, percentage records =0.33
Class label =1, percentage records =0.33
Class label =2, percentage records =0.33

Test data set class label distribution
=====
Class label =0, percentage records =0.33
Class label =1, percentage records =0.33
Class label =2, percentage records =0.33
```

Now, we have the class labels distributed uniformly between the test and train sets.

There's more...

We need to prepare the data carefully before its use in a machine learning algorithm. Providing a uniform class distribution to both the train and the test sets is key to building a successful classification model.

In practical machine learning scenarios, we create another dataset called as dev set in addition to the train and test sets. We may not get our model right in the first iteration. We don't want to show our test dataset to our model as this may bias our next iteration of model building. Hence, we create this dev set, which we can use as we iterate through our model building exercise.

The 80/20 rule of thumb that we specified in this recipe is an ideal scenario. However, in many practical applications, we may not have enough data to leave out that many instances for a test set. There are a few practical techniques, such as cross-validation, which come into play in such scenarios. In our next chapter, we will look at the various cross-validation techniques.

Finding the nearest neighbors

Before we jump into our recipe, let's spend some time understanding how to check if our classification model is performing to our satisfaction. In the introduction section, we introduced a term called confusion matrix.

A confusion matrix is a matrix arrangement of the actual versus the predicted class labels. Let's say we have a two-class problem, that is, our y can take either value, T or F. Let's say we trained a classifier to predict our y . In our test data, we know the actual value of y . We have the predicted value of our y from our model. Two values we can fill our confusion matrix, as follows:

		Predicted	
		T	F
Actual	T	TP	FN
	F	FP	TN

Here is a table where we conveniently list our results from the test set. Remember that we know the class labels in our test set; hence, we can compare our classification model output with the actual class label.

- ▶ Under TP, which is an abbreviation for True Positive, we have a count of all those records in the test set whose label is T, and where the model also predicted T
- ▶ Under FN, which is an abbreviation for False Negative, we have a count of all the records whose actual label is T, but the algorithm predicted N
- ▶ FP stands for False Positive, where the actual label is F, but the algorithm predicted it as T
- ▶ TN stands for True Negative, where the algorithm predicted both the label and the actual class label as F

With the knowledge of this confusion matrix, we can now derive performance metrics with which we can measure the quality of our classification model. In future chapters, we will explore more metrics, but for now, we will introduce the accuracy and error rate.

Accuracy is defined as the ratio of a correct prediction to the total number of predictions. From the confusion matrix, we know that the sum of TP and TN is the total number of correct predictions:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

Accuracy from the training set is always very optimistic. One should look at the test set's accuracy value to determine the true performance of the model.

Armed with this knowledge, let's jump into our recipe. The first classification algorithm that we will look at is K-Nearest Neighbor, in short, KNN. Before going into the details of KNN, let's look at a very simple classification algorithm, called the rote classifier algorithm. The rote classifier memorizes the entire training data, that is, it loads all the data in the memory. We need to perform classification on an unseen new training instance: it will try to match the new training instance with any of the training instances in the memory. It matches every attribute of the test instance with every attribute in the training instance. If it finds a match, it predicts the class label of the test instance as the class label of the matched training instance.

You should know by now that this classifier will fail if the test instance is not similar to any of the training instances loaded into the memory.

KNN is similar to rote classifier, except that instead of looking for an exact match, it uses a similarity measure. Similar to rote classifier, KNN loads all the training sets into the memory. When it needs to classify a test instance, it measures the distance between the test instance and all the training instances. Using this distance, it chooses K closest instances in the training set. Now, the prediction for the test set is based on the majority classes of the K nearest neighbors.

For example, if we have a two-class classification problem and we choose our K value as three, and if the given test record's three nearest neighbors have classes, 1, 1, and 0, it will classify the test instance as 1, which is the majority.

KNN belongs to a family of algorithms called instance-based learning. Additionally, as the decision to classify a test instance is taken last, it's also called a lazy learner.

Getting ready

For this recipe, we will generate some data using scikit's make_classification method. We will generate a matrix of four columns / attributes / features and 100 instances:

```
from sklearn.datasets import make_classification

import numpy as np
import matplotlib.pyplot as plt
import itertools

from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier

def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    x,y = make_classification(n_features=4)
```

```
return x,y

def plot_data(x,y):
    """
    Plot a scatter plot fo all variable combinations
    """
    subplot_start = 321
    col_numbers = range(0,4)
    col_pairs = itertools.combinations(col_numbers,2)

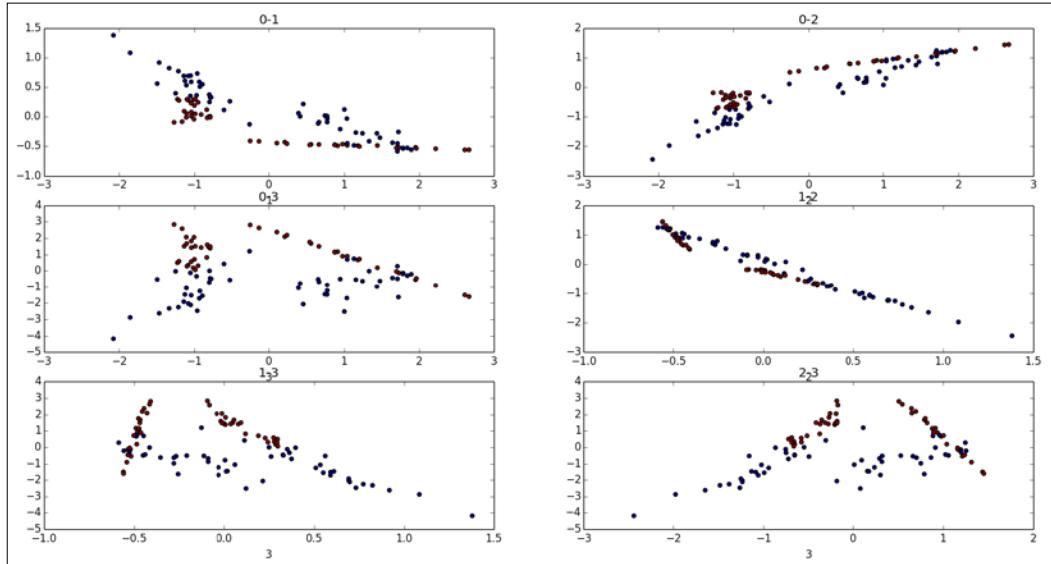
    for col_pair in col_pairs:
        plt.subplot(subplot_start)
        plt.scatter(x[:,col_pair[0]],x[:,col_pair[1]],c=y)
        title_string = str(col_pair[0]) + "-" + str(col_pair[1])
        plt.title(title_string)
        x_label = str(col_pair[0])
        y_label = str(col_pair[1])
        plt.xlabel(x_label)
        plt.ylabel(y_label)
        subplot_start+=1

    plt.show()

x,y = get_data()
plot_data(x,y)
```

The `get_data` function internally calls `make_classification` to generate test data for any classification task.

It's always good practice to visualize the data before starting to feed it into any algorithm. Our `plot_data` function produces a scatter plot between all the variables:



We have plotted all the variable combinations. The top two charts there in show combinations between the 0th and 1st column, followed by 0th and 2nd. The points are also colored by their class labels. This gives an idea of how much information is these variable combination to do a classification task.

How to do it...

We will separate our dataset preparation and model training into two different methods: `get_train_test` to get the train and test data, and `build_model` to build our model. Finally, we will use `test_model` to validate the usefulness of our model:

```
from sklearn.cross_validation import StratifiedShuffleSplit
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

def get_train_test(x,y):
    """
    Perpare a stratified train and test split
    """
    train_size = 0.8
    test_size = 1-train_size
    input_dataset = np.column_stack([x,y])
```

```
stratified_split = StratifiedShuffleSplit(input_dataset[:, -  
1], test_size=test_size, n_iter=1)  
  
for train_idx, test_idx in stratified_split:  
    train_x = input_dataset[train_idx, :-1]  
    train_y = input_dataset[train_idx, -1]  
    test_x = input_dataset[test_idx, :-1]  
    test_y = input_dataset[test_idx, -1]  
    return train_x, train_y, test_x, test_y  
  
def build_model(x,y,k=2):  
    """  
    Fit a nearest neighbour model  
    """  
    knn = KNeighborsClassifier(n_neighbors=k)  
    knn.fit(x,y)  
    return knn  
  
def test_model(x,y,knn_model):  
    y_predicted = knn_model.predict(x)  
    print classification_report(y,y_predicted)  
  
if __name__ == "__main__":  
  
    # Load the data  
    x,y = get_data()  
  
    # Scatter plot the data  
    plot_data(x,y)  
  
    # Split the data into train and test  
    train_x,train_y,test_x,test_y = get_train_test(x,y)  
  
    # Build the model  
    knn_model = build_model(train_x,train_y)  
  
    # Test the model  
    print "\nModel evaluation on training set"  
    print "=====\n"  
    test_model(train_x,train_y,knn_model)  
  
    print "\nModel evaluation on test set"  
    print "=====\n"  
    test_model(test_x,test_y,knn_model)
```

How it works...

Let's try to follow the code from our main method. We must start by calling `get_data` and plotting it using `plot_data`, as described in the previous section.

As mentioned previously, we need to separate a part of the training data for the testing that is required to evaluate our model. We then invoke the `get_train_test` method to do the same.

In `get_train_test`, we decide our train test split size, which is the standard 80/20. We then use 80 percent of our data to train our model. Now, we combine both `x` and `y` to a single matrix before the split using NumPy's `column_stack` method.

We then leverage `StratifiedShuffleSplit` discussed in the previous recipe in order to get a uniform class label distribution between our training and test sets.

Armed with our train and test sets, we are now ready to build our classifier. We must invoke the `build_model` with our training set, attributes `x`, and class labels `y`. This function also takes `K`, the number of neighbors, as a parameter, with a default value of two.

We use scikit-learn's KNN implementation, `KNeighborsClassifier`. We then create an object of the classifier and call the `fit` method to build our model.

We are ready to test how good the model is using our training data. We can pass our training data (`x` and `y`) and our model to the `test_model` function.

We know our actual class labels (`y`). We then invoke the `predict` function with our `x` to get the predicted labels. We then print some of the model evaluation metrics. We can start with printing the accuracy of the model, follow it up with a confusion matrix, and then finally show the output of a function called `classification_report`. scikit-learn's metrics module provides a function called `classification_report`, which can print several model evaluation metrics.

Let's look at our model metrics:

```
Model evaluation on training set
=====
Model accuracy = 91.25%

Confusion Matrix
=====
array([[40,  0],
       [ 7, 33]])
None

Classification Report
=====
      precision    recall  f1-score   support
0.0        0.85     1.00     0.92      40
1.0        1.00     0.82     0.90      40
avg / total     0.93     0.91     0.91      80
```

As you can see, our accuracy score is 91.25 percent. We will not repeat the definition of accuracy; you can refer to the introduction section.

Let's now look at our confusion matrix. The top left cell is the true positive cell. We can see that we have no false negatives but we have seven false positives (the first cell in the 2nd row).

Finally, we have precision, recall, an F1 score, and support in our classification report. Let's look at their definitions:

Precision is the ratio of the true positive and the sum of the true positive and false positive

Accuracy is the ratio of the true positive and the sum of the true positive and false negative

An F1 score is the harmonic mean of precision and sensitivity

We will see more about this metric in a separate recipe in a future chapter. For now, let's say we shall have high precision and recall values.

It is good to know that we have around 91 percent accuracy for our model, but the real test will be when it is run on the test data. Let's see the metrics for our test data:

```
Model evaluation on test set
=====
Model accuracy = 95.00%

Confusion Matrix
=====
array([[ 9,  1],
       [ 0, 10]])
None

Classification Report
=====
      precision    recall  f1-score   support
 0.0        1.00     0.90      0.95      10
 1.0        0.91     1.00      0.95      10

avg / total     0.95     0.95      0.95      20
```

It is good to know that our model has 95 percent accuracy for the test data, which is an indication of a good job in fitting the model.

There's more...

Let's look a little bit deeper into the model that we have built:

```
>>> pprint.pprint(knn_model.get_params())
{'algorithm': 'auto',
 'leaf_size': 30,
 'metric': 'minkowski',
 'n_neighbors': 2,
 'p': 2,
 'weights': 'uniform'}
```

We invoked a function called `get_params`. This function returns all the parameters that are passed to the model. Let's examine each of the parameters.

The first parameter refers to the underlying data structure used by the KNN implementation. As every record in the training set has to be compared against every other record, brute force implementation may be compute-resource heavy. Hence, we can choose either `kd_tree` or `ball_tree` as the data structure. A brute will use the brute force method of looping through all the records for every record.

Leaf size is the parameter that is passed to the `kd_tree` or `ball_tree` method.

Metric is the distance measure used to find the neighbors. The p-value of two reduces Minkowski to Euclidean distance.

Finally, we have the weights parameter. KNN decides the class label of the test instance based on the class label of its K nearest neighbors. A majority vote decides the class label for the test instance. However, if we set the weights to distance, then each neighbor is given a weight that is inversely proportional to its distance. Hence, in order to decide the class label of a test set, weighted voting is performed, rather than simple voting.

See also

- ▶ *Preparing data for model building* recipe in Chapter 6, Machine Learning I
- ▶ *Working with distance measures* recipe in Chapter 5, Data Mining - Finding a needle in a haystack

Classifying documents using Naïve Bayes

We will look at a document classification problem in this recipe. The algorithm that we will use is the Naïve Bayes classifier. The Bayes' rule is the engine powering the Naïve Bayes algorithm, as follows:

$$P(X|Y) = \frac{P(Y|X)*P(X)}{P(Y)}$$

It shows how likely it is for the event X to happen, given that we know event Y has already happened. Now, in our recipe, we will categorize or classify the text. Ours is a binary classification problem: given a movie review, we want to classify if the review is positive or negative.

In Bayesian terminology, we need to find the conditional probability: the probability that the review is positive given the review, and the probability that the review is negative given the review. Let's write it as an equation:

$$P(\text{class} = \text{positive} | \text{review}), \text{ and } P(\text{class} = \text{negative} | \text{review})$$

For any review, if we have the preceding two probability values, we can classify the review as positive or negative by comparing these values. If the conditional probability for negative is greater than the conditional probability for positive, we classify the review as negative, and vice versa.

Let's now discuss these probabilities using Bayes' rule:

$$P(\text{positive} | \text{review}) = \frac{P(\text{review} | \text{positive}) * P(\text{positive})}{P(\text{review})}$$

$$P(\text{negative} | \text{review}) = \frac{P(\text{review} | \text{negative}) * P(\text{negative})}{P(\text{review})}$$

As we are going to compare these two equations to finalize our prediction, we can ignore the denominator, which is a simple scaling factor.

The LHS (left-hand side) of the preceding equation is called the posterior probability.

Let's look at the numerator of the RHS (right-hand side):

$$P(\text{review} | \text{positive}) * P(\text{positive})$$

$P(\text{positive})$ is the probability of a positive class called the prior. It's our belief about the positive class label distribution based on our training set.

We will estimate it from our training test. It's calculated as follows:

$$P(\text{positive}) = \frac{\text{No of reviews with positive class label}}{\text{Total reviews in the corpus}}$$

$P(\text{review} | \text{positive})$ is the likelihood. It answers the question: what is the likelihood of getting the review, given that the class is positive. Again, we will estimate it from our training set.

Before we expand on the likelihood equation further, let's introduce the concept of independence assumption. The algorithm is prefixed as naïve because of this assumption. Contrary to the reality, we assume that the words appear in a document independent of each other. We will use this assumption to calculate the likelihood.

A review is a list of words. Let's put it in mathematical notation:

$$\text{review} = \{\text{word}_1, \text{word}_2, \dots, \text{word}_n\}$$

With the independence assumption, we can say that the probability of each of these words occurring together in a review is the product of all the individual probabilities of the constituent words in the review.

Now we can write the likelihood equation as follows:

$$P(\text{review} = \{\text{word}_1, \text{word}_2, \dots, \text{word}_n\} | \text{positive}) = \prod_{i=1}^n P(\text{word}_i | \text{positive})$$

So, given a new review, we can use these two equations, the prior and likelihood, to calculate whether the review is positive or negative.

Hopefully, you have followed till now. There is still a one last piece to the puzzle: how do we calculate the probability for the individual words?

$$P(\text{word}_i \mid \text{positive}) = \frac{\text{No of times word}_i \text{ occurs in reviews classified as positive}}{\text{Total number of words in all the reviews classified as positive}}$$

This step refers to training the model.

From our training set, we will take each review. We also know its label. For each word in this review, we will calculate the conditional probability and store it in a table. We can thus use these values to predict any future test instance.

Enough theory! Let's dive into our recipe.

Getting ready

For this recipe, we will use the NLTK library for both the data and the algorithm. During the installation of NLTK, we can also download the datasets. One such dataset is the movie review dataset. The movie review data is segregated into two categories, positive and negative. For each category, we have a list of words; the reviews are preseparated into words:

```
from nltk.corpus import movie_reviews
```

As shown here, we will include the datasets by importing the corpus module from NLTK.

We will leverage the `NaiveBayesClassifier` class, defined in NLTK, to build the model. We will pass our training data to a function called `train()` to build our model.

How to do it...

Let's start with importing the necessary function. We will follow it up with two utility functions. The first one retrieves the movie review data and the second one helps us split our data for the model into training and testing:

```
from nltk.corpus import movie_reviews
from sklearn.cross_validation import StratifiedShuffleSplit
import nltk
from nltk.corpus import stopwords
from nltk.collocations import BigramCollocationFinder
from nltk.metrics import BigramAssocMeasures

def get_data():
```

```
"""
Get movie review data
"""

dataset = []
y_labels = []
# Extract categories
for cat in movie_reviews.categories():
    # for files in each category
    for fileid in movie_reviews.fileids(cat):
        # Get the words in that category
        words = list(movie_reviews.words(fileid))
        dataset.append((words,cat))
        y_labels.append(cat)
return dataset,y_labels

def get_train_test(input_dataset,ylabels):
    """
    Prepare a stratified train and test split
    """

    train_size = 0.7
    test_size = 1-train_size
    stratified_split = StratifiedShuffleSplit(ylabels,test_size=test_size,n_iter=1,random_state=77)

    for train_indx,test_indx in stratified_split:
        train = [input_dataset[i] for i in train_indx]
        train_y = [ylabels[i] for i in train_indx]

        test = [input_dataset[i] for i in test_indx]
        test_y = [ylabels[i] for i in test_indx]
    return train,test,train_y,test_y
```

We will now introduce three functions, which are primarily feature-generating functions. We need to provide features or attributes to our classifier. These functions, given a review, generate a set of features from the review:

```
def build_word_features(instance):
    """
    Build feature dictionary
    Features are binary, name of the feature is word iteslf
    and value is 1. Features are stored in a dictionary
    called feature_set
    """

    # Dictionary to store the features
```

```
feature_set = {}
# The first item in instance tuple the word list
words = instance[0]
# Populate feature dicitonary
for word in words:
    feature_set[word] = 1
# Second item in instance tuple is class label
return (feature_set,instance[1])

def build_negate_features(instance):
    """
    If a word is preceeded by either 'not' or 'no'
    this function adds a prefix 'Not_' to that word
    It will also not insert the previous negation word
    'not' or 'no' in feature dictionary
    """
    # Retreive words, first item in instance tuple
    words = instance[0]
    final_words = []
    # A boolean variable to track if the
    # previous word is a negation word
    negate = False
    # List of negation words
    negate_words = ['no','not']
    # On looping through the words, on encountering
    # a negation word, variable negate is set to True
    # negation word is not added to feature dictionary
    # if negate variable is set to true
    # 'Not_' prefix is added to the word
    for word in words:
        if negate:
            word = 'Not_' + word
            negate = False
        if word not in negate_words:
            final_words.append(word)
        else:
            negate = True
    # Feature dictionary
    feature_set = {}
    for word in final_words:
        feature_set[word] = 1
    return (feature_set,instance[1])

def remove_stop_words(in_data):
```

```
"""
Utility function to remove stop words
from the given list of words
"""

stopword_list = stopwords.words('english')
negate_words = ['no', 'not']
# We dont want to remove the negate words
# Hence we create a new stop word list excluding
# the negate words
new_stopwords = [word for word in stopword_list if word not in
negate_words]
label = in_data[1]
# Remove stopw words
words = [word for word in in_data[0] if word not in new_stopwords]
return (words, label)

def build_keyphrase_features(instance):
    """
    A function to extract key phrases
    from the given text.
    Key Phrases are words of importance according to a measure
    In this key our phrase of is our length 2, i.e two words or
    bigrams
    """

    feature_set = {}
    instance = remove_stop_words(instance)
    words = instance[0]

    bigram_finder = BigramCollocationFinder.from_words(words)
    # We use the raw frequency count of bigrams, i.e. bigrams are
    # ordered by their frequency of occurence in descending order
    # and top 400 bigrams are selected.
    bigrams      = bigram_finder.nbest(BigramAssocMeasures.raw_
freq, 400)
    for bigram in bigrams:
        feature_set[bigram] = 1
    return (feature_set, instance[1])
```

Let's now write a function to build our model and later probe our model to find the usefulness of our model:

```
def build_model(features):
    """
    Build a naive bayes model
    with the gvien feature set.
    """
```

```
model = nltk.NaiveBayesClassifier.train(features)
return model

def probe_model(model,features,dataset_type = 'Train'):
    """
    A utility function to check the goodness
    of our model.
    """
    accuracy = nltk.classify.accuracy(model,features)
    print "\n" + dataset_type + " Accuracy = %0.2f"%(accuracy*100) +
    "%"

def show_features(model,no_features=5):
    """
    A utility function to see how important
    various features are for our model.
    """
    print "\nFeature Importance"
    print "=====\n"
    print model.show_most_informative_features(no_features)
```

It is very hard to get the model right at the first pass. We need to play around with different features, and parameter tuning. This is mostly a trial and error process. In the next section of code, we will show our different passes by improving our model:

```
def build_model_cycle_1(train_data,dev_data):
    """
    First pass at trying out our model
    """
    # Build features for training set
    train_features = map(build_word_features,train_data)
    # Build features for test set
    dev_features = map(build_word_features,dev_data)
    # Build model
    model = build_model(train_features)
    # Look at the model
    probe_model(model,train_features)
    probe_model(model,dev_features,'Dev')

    return model

def build_model_cycle_2(train_data,dev_data):
    """
    Second pass at trying out our model
    """
```

```
# Build features for training set
train_features = map(build_negate_features, train_data)
# Build features for test set
dev_features = map(build_negate_features, dev_data)
# Build model
model = build_model(train_features)
# Look at the model
probe_model(model, train_features)
probe_model(model, dev_features, 'Dev')

return model

def build_model_cycle_3(train_data, dev_data):
    """
    Third pass at trying out our model
    """

    # Build features for training set
    train_features = map(build_keyphrase_features, train_data)
    # Build features for test set
    dev_features = map(build_keyphrase_features, dev_data)
    # Build model
    model = build_model(train_features)
    # Look at the model
    probe_model(model, train_features)
    probe_model(model, dev_features, 'Dev')
    test_features = map(build_keyphrase_features, test_data)
    probe_model(model, test_features, 'Test')

    return model
```

Finally, we will write a code with which we can invoke all our functions that were defined previously:

```
if __name__ == "__main__":
    # Load data
    input_dataset, y_labels = get_data()
    # Train data
    train_data, all_test_data, train_y, all_test_y = get_train_
    test(input_dataset, y_labels)
    # Dev data
    dev_data, test_data, dev_y, test_y = get_train_test(all_test_
    data, all_test_y)
```

```
# Let us look at the data size in our different
# datasets
print "\nOriginal Data Size = ", len(input_dataset)
print "\nTraining Data Size = ", len(train_data)
print "\nDev Data Size = ", len(dev_data)
print "\nTesting Data Size = ", len(test_data)

# Different passes of our model building exercise
model_cycle_1 = build_model_cycle_1(train_data, dev_data)
# Print informative features
show_features(model_cycle_1)
model_cycle_2 = build_model_cycle_2(train_data, dev_data)
show_features(model_cycle_2)
model_cycle_3 = build_model_cycle_3(train_data, dev_data)
show_features(model_cycle_3)
```

How it works...

Let's try to follow this recipe from the main function. We started with invoking the `get_data` function. As explained before, the movie review data is stored as two categories, positive and negative. Our first loop goes through these categories. With these categories, we retrieved the file IDs for these categories in the second loop. Using these file IDs, we retrieve the words, as follows:

```
words = list(movie_reviews.words(fileid))
```

We will append these words to a list called `dataset`. The class label is appended to another list called `y_labels`.

Finally, we return the words and corresponding class labels:

```
return dataset, y_labels
```

Equipped with the dataset, we need to divide this dataset into the test and the train datasets:

```
# Train data
train_data, all_test_data, train_y, all_test_y = get_train_
test(input_dataset, y_labels)
```

We invoked the `get_train_test` function with an input dataset and the class labels. This function provides us with a stratified sample. We are using 70 percent of our data for the training set and the rest for the test set.

Once again, we invoke `get_train_test` with the test dataset returned from the previous step:

```
# Dev data
dev_data, test_data, dev_y, test_y = get_train_test(all_test_
data, all_test_y)
```

We created a separate dataset and called it the dev dataset. We need this dataset to tune our model. We want our test set to really behave as a test set. We don't want to expose our test set during the different passes of our model building exercise.

Let's print the size of our train, dev, and test datasets:

Original Data Size = 2000
Training Data Size = 1400
Dev Data Size = 420
Testing Data Size = 180

As you can see, 70 percent of the original data is assigned to our training set. We have again split the rest 30 percent into a 70/30 percent split for Dev and Testing.

Let's start our model building activity. We will call `build_model_cycle_1` with our training and dev datasets. In this function, we will first create our features by calling `build_word_feature` using a map on all the instances in our dataset. The `build_word_feature` is a simple feature-generating function. Every word is a feature. The output of this function is a dictionary of features, where the key is the word itself and the value is one. These types of features are typically called Bag of Words (BOW). The `build_word_features` is invoked using both the training and the dev data:

```
# Build features for training set
train_features = map(build_negate_features,train_data)
# Build features for test set
dev_features = map(build_negate_features,dev_data)
```

We will now proceed to train our model with the generated feature:

```
# Build model
model = build_model(train_features)
```

We need to test how good our model is. We use the `probe_model` function to do this. `Probe_model` takes three parameters. The first parameter is the model of interest, the second parameter is the feature against which we want to see how good our model is, and the last parameter is a string used for display purposes. The `probe_model` function calculates the accuracy metric using the `accuracy` function in the `nltk.classify` module.

We invoke `probe_model` twice: once with the training data to see how good the model is on our training dataset, and then once with our dev dataset:

```
# Look at the model
probe_model(model,train_features)
probe_model(model,dev_features,'Dev')
```

Let's now look at the accuracy figures:

```
Train Accuracy = 98.07%
```

```
Dev Accuracy = 69.76%
```

Our model is behaving very well using the training data. This is not surprising as the model has already seen it during the training phase. It's doing a good job at classifying the training record correctly. However, our dev accuracy is very poor. Our model is able to classify only 60 percent of the dev instances correctly. Surely our features are not informative enough to help our model classify the unseen instances with a good accuracy. It will be good to see which features are contributing more towards discriminating a review into positive and negative:

```
show_features(model_cycle_1)
```

We will invoke the `show_features` function to look at the features' contribution towards the model. The `Show_features` function utilizes the `show_most_informative_feature` function from the NLTK classifier object. The most important features in our first model are as follows:

Feature Importance			
=====			
Most Informative Features			
stupidity = 1	neg : pos =	15.0 : 1.0	
unconvincing = 1	neg : pos =	12.3 : 1.0	
wonderfully = 1	pos : neg =	11.4 : 1.0	
outstanding = 1	pos : neg =	11.0 : 1.0	
warned = 1	neg : pos =	10.3 : 1.0	

The way to read it is: the feature `stupidity = 1` is 15 times more effective for classifying a review as negative.

Let's now do a second round of building this model using a new set of features. We will do this by invoking `build_model_cycle_2`. `build_model_cycle_2` is very similar to `build_model_cycle_1` except for the feature generation function called inside map function.

The feature generation function is called `build_negate_features`. Typically, words such as not and no are called negation words. Let's assume that our reviewer says that the movie is not good. If we use our previous feature generator, the word good would be treated equally in both the positive and negative reviews. We know that the word good should be used to discriminate the positive reviews. To avoid this problem, we will look for the negation words no and not in our word list. We want to modify our example sentence as follows:

```
"movie is not good" to "movie is not_good"
```

This way, `no_good` can be used as a good feature to discriminate the negative reviews from the positive reviews. The `build_negate_features` function does this job.

Let's now look at our probing output for the model built with this negation feature:

```
Train Accuracy = 98.36%
Dev Accuracy = 70.24%
```

We improved our model accuracy on our dev data by almost 2 percent. Let's now look at the most informative features for this model:

Feature Importance			
=====			
Most Informative Features			
<code>stupidity</code> = 1	<code>neg : pos</code>	=	<code>15.0 : 1.0</code>
<code>wonderfully</code> = 1	<code>pos : neg</code>	=	<code>14.7 : 1.0</code>
<code>unconvincing</code> = 1	<code>neg : pos</code>	=	<code>12.3 : 1.0</code>
<code>Not_funny</code> = 1	<code>neg : pos</code>	=	<code>11.7 : 1.0</code>
<code>outstanding</code> = 1	<code>pos : neg</code>	=	<code>11.0 : 1.0</code>

Look at the last feature. Adding negation to funny, the '`Not_funny`' feature is 11.7 times more informative for discriminating a review as negative.

Can we do better on our model accuracy ? Currently, we are at 70 percent. Let's do a third run with a new set of features. We will do this by invoking `build_model_cycle_3`. `build_model_cycle_3` is very similar to `build_model_cycle_2` except for the feature generation function called inside map function.

The `build_keyphrase_features` function is used as a feature generator. Let's look at the function in detail. Instead of using the words as features, we will generate key phrases from the review and use them as features. Key phrases are phrases that we consider important using some metric. Key phrases can be made of either two, three, or n words. In our case, we will use two words (bigrams) to build our key phrase. The metric that we will use is the raw frequency count of these phrases. We will choose the phrases whose frequency count is higher. We will do some simple preprocessing before generating our key phrases. We will remove all the stopwords and punctuation from our word list. The `remove_stop_words` function is invoked to remove the stopwords and punctuation. NLTK's corpus module has a list of English stopwords. We can retrieve it as follows:

```
stopword_list = stopwords.words('english')
```

Similarly, the string module in Python maintains a list of punctuation. We will remove the stopwords and punctuation as follows:

```
words = [word for word in in_data[0] if word not in new_stopwords \
and word not in punctuation]
```

However, we will not remove `not` and `no`. We will create a new set of stopwords by not, including the negation words in the previous step:

```
new_stopwords = [word for word in stopword_list if word not in negate_words]
```

We will leverage the `BigramCollocationFinder` class from NLTK to generate our key phrases:

```
bigram_finder = BigramCollocationFinder.from_words(words)
# We use the raw frequency count of bigrams, i.e. bigrams are
# ordered by their frequency of occurrence in descending order
# and top 400 bigrams are selected.
bigrams      = bigram_finder.nbest(BigramAssocMeasures.raw_
freq, 400)
```

Our metric is the frequency count. You can see that we specified it as `raw_freq` in the last line. We will ask the collocation finder to return us a maximum of 400 phrases.

Loaded with our new feature, we will proceed to build our model and test the correctness of our model. Let's look at the output of our model:

Train Accuracy = 100.00%

Dev Accuracy = 80.00%

Yes! We have achieved a great deal of improvement on our dev set. From 68 percent accuracy in our first pass with word features, we have moved from 12 percent up to 80 percent with our key phrase features. Let's now expose our test set to this model and check the accuracy:

```
test_features = map(build_keyphrase_features,test_data)
probe_model(model,test_features,'Test')
```

Test Accuracy = 84.44%

Our test set's accuracy is greater than our dev set's accuracy. We did a good job in training a good model that works well on an unseen dataset. Before we end this recipe, let's look at the key phrases that are the most informative:

Feature Importance		
=====		
Most Informative Features		
(u'waste', u'time') = 1	neg : pos =	12.2 : 1.0
(u'oscar', u'nomination') = 1	pos : neg =	10.3 : 1.0
(u'one', u'worst') = 1	neg : pos =	10.2 : 1.0
(u'works', u'well') = 1	pos : neg =	9.7 : 1.0
(u'low', u'key') = 1	pos : neg =	9.7 : 1.0

The key phrase, Oscar nomination, is 10 times more helpful in discriminating a review as positive. You can't deny this. We can see that our key phrases are very informative, and hence, our model performed better than the previous two runs.

There's more...

How did we know that 400 key phrases and the metric frequency count is the best parameter for bigram generation? Trial and error. Though we didn't list our trial and error process, we pretty much ran it with various combinations such as 200 phrases with pointwise mutual information, and similar other methods.

This is what needs to be done in the real world. However, instead of a blind search through the parameter space every time, we looked at the most informative features. This gave us a clue on the discriminating power of the features.

See also

- ▶ *Preparing data for model building* recipe in Chapter 6, Machine Learning I

Building decision trees to solve multiclass problems

In this recipe, we will look at building decision trees to solve multiclass classification problems. Intuitively, a decision tree can be defined as a process of arriving at an answer by asking a series of questions: a series of if-then statements arranged hierarchically forms a decision tree. Due to this nature, they are easy to understand and interpret.

Refer to the following link for a detailed introduction to decision trees:

https://en.wikipedia.org/wiki/Decision_tree

Theoretically, many decision trees can be built for a given dataset. Some of the trees are more accurate than others. There are efficient algorithms for developing a reasonably accurate tree in a limited time. One such algorithm is Hunt's algorithm. Algorithms such as ID3, C4.5, and CART (Classification and Regression Trees) are based on Hunt's algorithm. Hunt's algorithm can be outlined as follows:

Given a dataset, D, of n records and with each record having m attributes / features / columns and each record labeled either y1, y2, or y3, the algorithm proceeds as follows:

- ▶ If all the records in D belong to the same class label, say y1, then y1 is the leaf node of the tree and is labeled y1.

- If D has records that belong to more than one class label, a feature test condition is employed to divide the records into smaller subsets. Let's say in the initial run, we run a feature test condition on all the attributes and find a single attribute that is able to split the datasets into three smaller subsets. This attribute becomes the root node. We apply the test condition on all the three subsets to figure out the next level of nodes. This process is performed iteratively.

Notice that when we defined our classification, we defined three class labels, y_1 , y_2 , and y_3 . This is different from the problems that we solved in the previous two recipes, where we had only two labels. This is a multiclass problem. Our Iris dataset that we used in most of our recipes is a three-class problem. We had our records distributed across three class labels. We can generalize it to an n-class problem. Digit recognition is another example by which we need to classify a given image in one of the digits between zero and nine. Many real-word problems are inherently multiclass. Some algorithms are also inherently capable of handling multiclass problems. No change is required for these algorithms. The algorithms that we will discuss in the chapters are all capable of handling multiclass problems. Decision trees, Naïve Bayes, and KNN algorithms are good at handling multiclass problems.

Let's see how we can leverage decision trees to handle multiclass problems in this recipe. It also helps to have a good understanding of decision trees. Random forest, which we will venture into in the next chapter, is a more sophisticated method used widely in the industry and is based on decision trees.

Let's now dive into our decision tree recipe.

Getting ready

We will use the Iris dataset to demonstrate how to build decision trees. Decision trees are a non-parametric supervised learning method that can be used to solve both classification and regression problems. As explained previously, the advantages of using decision trees are manifold, as follows:

- They are easily interpretable
- They require very little data preparation and data-to-feature conversion: remember our feature generation methods in the previous recipe
- They naturally support multiclass problems

Decision trees are not without problems. Some of the problems that they pose are as follows:

- They can easily overfit: a high accuracy in a training set and very poor performance with a test data.
- There can be millions of trees that can be fit to a given dataset.

- ▶ The class imbalance problem may affect decision trees heavily. The class imbalance problem arises when our training set does not consist of an equal number of instances for both the class labels in a binary classification problem. This applies to multiclass problems as well.

An important part of decision trees is the feature test condition. Let's spend some time understanding the feature test condition. Typically, each attribute in our instance can be either understood.

Binary attribute: This is where an attribute can take two possible values, for example, true or false. The feature test condition should return two values in this case.

Nominal attribute: This is where an attribute can take more than two values, for example, n values. The feature test condition should either output n output or group them into binary splits.

Ordinal attribute: This is where an implicit order in their values exists. For example, let's take an imaginary attribute called size, which can take on the values small, medium, or large. There are three values that the attribute can take and there is an order for them: small, medium, large. They are handled by the feature attribute test condition that is similar to the nominal attribute.

Continuous attributes: These are attributes that can take continuous values. They are discretized into ordinal attributes and then handled.

A feature test condition is a way to split the input records into subsets based on a criteria or metric called impurity. This impurity is calculated with respect to the class label for each attribute in the instance. The attribute contributing to the highest impurity is chosen as the data splitting attribute, or in other words, the node for that level in the tree.

Let's see an example to explain it. We will use a measure called entropy to calculate the impurity.

Entropy is defined as follows:

$$E(X) = - \sum_{i=1}^n P(x_i) \log_2(P(x_i))$$

Where:

$$P(x_i) = \frac{\text{Count of } x_i}{\text{length}(X)}$$

Let's consider an example:

```
X = {2, 2}
```

We can now calculate the entropy for this set, as follows:

$$\begin{aligned} & -\left(\frac{2}{2} * \log_2\left(\frac{2}{2}\right) + \frac{2}{2} * \log_2\left(\frac{2}{2}\right)\right) \\ & -(1 * \log_2(1) + 1 * \log_2(1)) = 0.0 \end{aligned}$$

The entropy for this set is 0. An entropy of 0 indicates homogeneity. It is very easy to code entropy in Python. Look at the following code list:

```
import math

def prob(data, element):
    """Calculates the percentage count of a given element

    Given a list and an element, returns the elements percentage count

    """
    element_count = 0
    # Test condition to check if we have proper input
    if len(data) == 0 or element == None \
            or not isinstance(element, (int, long, float)):
        return None
    element_count = data.count(element)
    return element_count / (1.0 * len(data))

def entropy(data):
    """Calculate entropy

    """
    entropy = 0.0

    if len(data) == 0:
        return None

    if len(data) == 1:
        return 0
    try:
        for element in data:
            p = prob(data, element)
```

```
    entropy+=-1*p*math.log(p,2)
except ValueError as e:
    print e.message

return entropy
```

For the purpose of finding the best splitting variable, we will leverage the entropy. First, we will calculate the entropy based on the class labels, as follows:

$$Entropy(t) = -\sum_{i=0}^{c-1} p(i|t) \log_2 p(i|t)$$

Let's define another term called information gain. Information gain is a measure to find which attribute in the given instance is most useful for discrimination between the class labels.

Information gain is the difference between an entropy of the parent and an average entropy of the child nodes. At each level in the tree, we will use information gain to build the tree:

https://en.wikipedia.org/wiki/Information_gain_in_decision_trees

We will start with all the attributes in a training set and calculate the overall entropy. Let's look at the following example:

Lead Actor	Oscar Winning	Box Office	Watch
Y	Y	N	Y
Y	N	Y	N
N	N	Y	Y
N	Y	Y	Y

The preceding dataset is imaginary data collected for a user to figure out what kind of movies he is interested in. There are four attributes: the first one is about whether the user watches a movie based on the lead actor, the second attribute is about if the user makes his decision to watch the movie based on whether or not it won an Oscar, and the third one is about if the user decides to watch a movie based on whether or not it is a box office success.

In order to build a decision tree for the preceding example, we will start with the entropy calculation of the whole dataset. This is a two-class problem, hence $c = 2$. Additionally, there are a total of four records, hence, the entropy of the whole dataset is as follows:

$$E(D) = -\left(\frac{1}{4} * \log_2 \left(\frac{1}{4} \right) + \frac{3}{4} * \log_2 \left(\frac{3}{4} \right) \right)$$

The overall entropy of the dataset is 0.811.

Now, let's look at the first attribute, the lead attribute. For a lead actor, Y, there is one instance class label that says Y and another one that says N. For a lead actor, N, both the instance class labels are N. We will calculate the average entropy as follows:

```
entropy_lead_actor_Y = 2/4.0 * -(1/2.0 * log(1/2.0,2) + 1/2.0 * log(1/2.0,2))
entropy_lead_actor_N = 2/4.0 * -(0 + 2/2.0 * log(2/2.0,2))
entrop_lead_actor = entropy_lead_actor_Y + entropy_lead_actor_N
```

It's an average entropy. There are two records with a lead actor as Y and two records with lead actors as N; hence, we have $2/4 \cdot 0$ multiplied to the entropy value.

As the entropy is calculated for this subset of data, we can see that out of the two records, one of them has a class label of Y and another one has a class label of N for the lead actor Y. Similarly, for the lead actor N, both the records have a class label of N. Thus, we get the average entropy for this attribute.

The average entropy value for the lead actor attribute is 0.5.

The information gain is now $0.811 - 0.5 = 0.311$.

Similarly, we will find the information gain for all the attributes. The attribute with the highest information gain wins and becomes the root node of the decision tree.

The same process is repeated in order to find the second level of the nodes, and so on.

How to do it...

Let's load the required libraries. We will follow it with two functions, one to load the data and the second one to split the data into a training set and a test it:

```
from sklearn.datasets import load_iris
from sklearn.cross_validation import StratifiedShuffleSplit
import numpy as np
from sklearn import tree
from sklearn.metrics import accuracy_score, classification_
report, confusion_matrix
import pprint

def get_data():
    """
    Get Iris data
    """
    data = load_iris()
    x = data['data']
```

```
y = data['target']
label_names = data['target_names']

return x,y,label_names.tolist()

def get_train_test(x,y):
    """
    Perpare a stratified train and test split
    """
    train_size = 0.8
    test_size = 1-train_size
    input_dataset = np.column_stack([x,y])
    stratified_split = StratifiedShuffleSplit(input_dataset[:, -1], \
        test_size=test_size,n_iter=1,random_state = 77)

    for train_idx,test_idx in stratified_split:
        train_x = input_dataset[train_idx,:-1]
        train_y = input_dataset[train_idx,-1]
        test_x = input_dataset[test_idx,:-1]
        test_y = input_dataset[test_idx,-1]
    return train_x,train_y,test_x,test_y
```

Let's write the functions to help us build and test the decision tree model:

```
def build_model(x,y) :
    """
    Fit the model for the given attribute
    class label pairs
    """
    model = tree.DecisionTreeClassifier(criterion="entropy")
    model = model.fit(x,y)
    return model

def test_model(x,y,model,label_names) :
    """
    Inspect the model for accuracy
    """
    y_predicted = model.predict(x)
    print "Model accuracy = %0.2f"%(accuracy_score(y,y_predicted) * 100) + "%\n"
    print "\nConfusion Matrix"
    print "===="
    print pprint.pprint(confusion_matrix(y,y_predicted))
```

```
print "\nClassification Report"
print "===="

print classification_report(y,y_predicted,target_names=label_
names)
```

Finally, the main function to invoke all the other functions that we defined is as follows:

```
if __name__ == "__main__":
    # Load the data
    x,y,label_names = get_data()
    # Split the data into train and test
    train_x,train_y,test_x,test_y = get_train_test(x,y)
    # Build model
    model = build_model(train_x,train_y)
    # Evaluate the model on train dataset
    test_model(train_x,train_y,model,label_names)
    # Evaluate the model on test dataset
    test_model(test_x,test_y,model,label_names)
```

How it works...

Let's start with the main function. We invoke `get_data` in the `x`, `y`, and `label_names` variables in order to retrieve the Iris dataset. We took the label names so that when we see our model accuracy, we can measure it by individual labels. As said previously, the Iris data poses a three-class problem. We will need to build a classifier that can classify any new instances in one of the tree types: setosa, versicolor, or virginica.

Once again, as in the previous recipes, `get_train_test` returns stratified train and test datasets. We then leverage `StratifiedShuffleSplit` from scikit-learn to get the training and test datasets with an equal class label distribution.

We must invoke the `build_model` method to induce a decision tree on our training set. The `DecisionTreeClassifier` class in the model tree of scikit-learn implements a decision tree:

```
model = tree.DecisionTreeClassifier(criterion="entropy")
```

As you can see, we specified that our feature test condition is an entropy using the `criterion` variable. We then build the model by calling the `fit` function and return the model to the calling program.

Now, let's proceed to evaluate our model by using the `test_model` function. The model takes instances `x`, class labels `y`, decision tree model `model`, and the name of the class labels `label_names`.

The module `metric` in scikit-learn provides three evaluation criteria:

```
from sklearn.metrics import accuracy_score, classification_
                           report, confusion_matrix
```

We defined accuracy in the previous recipe and the introduction section.

A confusion matrix prints the confusion matrix defined in the introduction section. A confusion matrix is a good way of evaluating the model performance. We are interested in the cell values having true positive and false positive values.

Finally, we also have `classification_report` to print the precision, recall, and F1 score.

We must evaluate the model on the training data first:

```
Model accuracy = 100.00%

Confusion Matrix
=====
array([[40,  0,  0],
       [ 0, 40,  0],
       [ 0,  0, 40]])
None
[[40  0  0]
 [ 0 40  0]
 [ 0  0  0]]

Classification Report
=====
      precision    recall   f1-score   support
setosa      1.00     1.00     1.00      40
versicolor  1.00     1.00     1.00      40
virginica   1.00     1.00     1.00      40
avg / total  1.00     1.00     1.00     120
```

We have done a great job with the training dataset. We have 100 percent accuracy. The true test is with the test dataset where the rubber meets the road.

Let's look at the model evaluation using the test dataset:

```
Model accuracy = 100.00%

Confusion Matrix
=====
array([[10,  0,  0],
       [ 0, 10,  0],
       [ 0,  0, 10]])
None
[[10  0  0]
 [ 0 10  0]
 [ 0  0  0]]

Classification Report
=====
precision    recall   f1-score   support
setosa       1.00     1.00      1.00      10
versicolor   1.00     1.00      1.00      10
virginica    1.00     1.00      1.00      10
avg / total  1.00     1.00      1.00      30
```

Our classifier has performed extremely well with the test set as well.

There's more...

Let's probe the model to see how the various features contributed towards discriminating the classes.

```
def get_feature_names():
    data = load_iris()
    return data['feature_names']

def probe_model(x,y,model,label_names):

    feature_names = get_feature_names()
    feature_importance = model.feature_importances_
    print "\nFeature Importance\n"
    print "=====\\n"
    for i,feature_name in enumerate(feature_names):
        print "%s = %0.3f"%(feature_name,feature_importance[i])

    # Export the decision tree as a graph
    tree.export_graphviz(model,out_file='tree.dot')
```

The tree classifier object provides an attribute called `feature_importances_`, which can be called to retrieve the importance of the various features towards building our model.

We wrote a simple function, `get_feature_names`, in order to retrieve the names of our attributes. However, this can be added as a part of `get_data`.

Let's look at the print statement output:

```
Feature Importance
=====
sepal length (cm) = 0.014
sepal width (cm) = 0.014
petal length (cm) = 0.074
petal width (cm) = 0.897
```

This looks as if the petal width and petal length are contributing more towards our classification.

Interestingly, we can also export the tree built by the classifier as a dot file and it can be visualized using the GraphViz package. In the last line, we export our model as a dot file:

```
# Export the decision tree as a graph
tree.export_graphviz(model,out_file='tree.dot')
```

You can download and install the Graphviz package to visualize this:

<http://www.graphviz.org/>

See also

- ▶ *Preparing data for model building* recipe in Chapter 6, Machine Learning I
- ▶ *Finding nearest neighbors* recipe in Chapter 6, Machine Learning I
- ▶ *Classifying documents using Naive Bayes* recipe in Chapter 6, Machine Learning I

7

Machine Learning 2

In this chapter, we will cover the following recipes:

- ▶ Predicting real-valued numbers using regression
- ▶ Learning regression with L2 shrinkage – ridge
- ▶ Learning regression with L1 shrinkage – LASSO
- ▶ Using cross-validation iterators with L1 and L2 shrinkage

Introduction

In this chapter, we will introduce regression techniques and how they can be coded in Python. We will follow it up with a discussion on some of the drawbacks that are inherent with regression methods, and discuss how to address the same using shrinkage methods. There are some parameters that need to be set in the shrinkage methods. We will discuss cross-validation techniques to find the optimal parameter values for the shrinkage methods.

We saw classification problems in the previous chapter. In this chapter, let's turn our attention towards regression problems. In classification, the response variable y was either binary or a set of discrete values (in the case of multiclass and multilabel problems). In contrast, the response variable in regression is a real-valued number.

Regression can be thought of as a function approximation. The job of regression is to find a function such that when X , a set of random variables, is provided as an input to that function, it should return y , the response variable. X is also referred to as an independent variable and y is referred as a dependent variable.

We will leverage the techniques that we learnt in the previous chapter to divide our dataset into train, dev, and test sets, build our model iteratively on the train set, and validate it on dev. Finally, we will use our test set to get a good picture of the goodness of our model.

We will start the chapter with a recipe for simple linear regression using the least square estimation. At the beginning of the first recipe, we will provide a crisp introduction to the framework of regression, which is essential background information required to understand the other recipes in this chapter. Though very powerful, the simple regression framework suffers from a drawback. As there is no control over the upper and lower limits on the values that the coefficients of linear regression can take, they tend to overfit the given data. (The cost equation of linear regression is unconstrained. We will discuss more about it in the first recipe). The output regression model may not perform very well on unseen datasets. Shrinkage methods are used to address this problem. Shrinkage methods are also called regularization methods. In the next two recipes, we will cover two different shrinkage methods called LASSO and ridge. In our final recipe, we will introduce the concept of cross-validation and see how we can use it to our advantage in estimating the parameter, alpha, that is passed to ridge regression, a type of shrinkage method.

Predicting real-valued numbers using regression

Before we delve into this recipe, let's quickly understand how regression generally operates. This introduction is essential for understanding this and subsequent recipes.

Regression is a special form of function approximation. Here are the set of predictors:

$$X = \{x_1, x_2, \dots, x_n\}$$

With each instance, x_i has m attributes:

$$x_i = \{x_{i1}, x_{i2}, \dots, x_{im}\} \text{ where } i = 1 \text{ to } n$$

The job of regression is to find a function such that when X is provided as an input to that function, it should return a Y response variable. Y is a vector of real-valued entries:

$$F(X) = Y$$

We will use the Boston housing dataset in order to explain the regression framework.

The following link provides a good introduction to the Boston housing dataset:

<https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names>.

The response variable, y in this case, is the median value of an owner-occupied home in the Boston area. There are 13 predictors. The preceding web link provides a good description of all the predictor variables. The regression problem is defined as finding a function, F , such that if we give a previously unseen predictor value to this function, it should be able to give us the median house price.

The function, $F(x)$, which is the output of our linear regression model, is a linear combination of the input, x , hence the name linear regression:

$$F(X) = w_0 + w_1x_1 + w_2x_2 + \dots + w_mx_m$$

The w_i variable is the unknown value in the preceding equation. The modeling exercise is about discovering the w_i variable. Using our training data, we will find the value of w_i ; w_i is called the coefficient of the regression model.

A linear regression modeling problem is framed as: using the training data to find the coefficients:

$$w = (w_0, w_1, w_2, \dots, w_m)$$

Such that:

$$\sum_{i=1}^n \left(y_i - \sum_{j=0}^m x_j w_j \right)^2$$

The above formula is as low as possible.

The lower the value of this equation (called the cost function in optimization terminology), the better the linear regression model. So, the optimization problem is to minimize the preceding equation, that is, find the values for the w_i coefficient so that it minimizes the equation. We will not delve into the details of the optimization routines that are used. However, it is good to know this objective function because the next two recipes expect you to understand it.

Now, the question is how do we know that the model that we built using the training data, that is, our newly found coefficients, w_1, w_2, \dots, w_m are good enough to accurately predict unseen records? Once again, we will leverage the cost function defined previously. When we apply the model in our dev set or test set, we find the average square of the difference between the actual and predicted values, as follows:

$$\frac{1}{n} \sum_{i=1}^n (y_{true} - y_{predicted})^2$$

The preceding equation is called the mean squared error—the metric by which we can say that our regression model is worthy of use. We want an output model where the average square of the difference between the actual and predicted values is as low as possible. This method of finding the coefficients is called the least square estimation.

We will use scikit-learn's `LinearRegression` class. However, it internally uses the `scipy.linalg.lstsq` method. The method of least squares provides us with a closed form solution to the regression problem. Refer to the following links for more information on the method of least squares and the derivation for the least squares:

https://en.wikipedia.org/wiki/Least_squares.

[https://en.wikipedia.org/wiki/Linear_least_squares_\(mathematics\)](https://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)).

We gave a very simple introduction to regression. Curious readers can refer to the following books <http://www.amazon.com/exec/obidos/ASIN/0387952845/trevorhastie-20>

<http://www.amazon.com/Neural-Networks-Learning-Machines-Edition/dp/0131471392>

Getting ready

The Boston data has 13 attributes and 506 instances. The target variable is a real number and the median value of the houses is in the thousands.

Refer to the following UCI link for more information about the Boston dataset: <https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names>.

We will provide the names of these predictor and response variables, as follows:

1. CRIM	per capita crime rate by town
2. ZN	proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS	proportion of non-retail business acres per town
4. CHAS	Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. NOX	nitric oxides concentration (parts per 10 million)
6. RM	average number of rooms per dwelling
7. AGE	proportion of owner-occupied units built prior to 1940
8. DIS	weighted distances to five Boston employment centres
9. RAD	index of accessibility to radial highways
10. TAX	full-value property-tax rate per \$10,000
11. PTRATIO	pupil-teacher ratio by town
12. B	$1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
13. LSTAT	% lower status of the population
14. MEDV	Median value of owner-occupied homes in \$1000's

How to do it...

We will start with loading all the necessary libraries. We will follow it up by defining our first function, `get_data()`. In this function, we will read the Boston dataset and return it as predictor `x` and response variable `y`:

```
# Load libraries
from sklearn.datasets import load_boston
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
def get_data():
    """
    Return boston dataset
    as x - predictor and
    y - response variable
    """
    data = load_boston()
    x = data['data']
    y = data['target']
    return x,y
```

In our `build_model` function, we will construct our linear regression model with the given data. The following two functions, `view_model` and `model_worth`, are used to introspect the model that we have built:

```
def build_model(x,y):
    """
    Build a linear regression model
    """
    model = LinearRegression(normalize=True,fit_intercept=True)
    model.fit(x,y)
    return model

def view_model(model):
    """
    Look at model coefficients
    """
    print "\n Model coefficients"
    print "=====\\n"
    for i,coef in enumerate(model.coef_):
        print "\tCoefficient %d %0.3f"%(i+1,coef)

    print "\\n\tIntercept %0.3f"%(model.intercept_)

def model_worth(true_y,predicted_y):
    """
    Evaluate the model
    """
    print "\tMean squared error = %0.2f"%(mean_squared_error(true_y,predicted_y))
```

The `plot_residual` function is used to plot the errors in our regression model:

```
def plot_residual(y,predicted_y):
    """
    Plot residuals
    """
    plt.cla()
    plt.xlabel("Predicted Y")
    plt.ylabel("Residual")
    plt.title("Residual Plot")
    plt.figure(1)
    diff = y - predicted_y
    plt.plot(predicted_y,diff,'go')
    plt.show()
```

Finally, we will write our `main` function, which is used to invoke all the preceding functions:

```
if __name__ == "__main__":  
  
    x,y = get_data()  
  
    # Divide the data into Train, dev and test  
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_size = 0.3,random_state=9)  
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_all,test_size=0.3,random_state=9)  
  
    # Build the model  
    model = build_model(x_train,y_train)  
    predicted_y = model.predict(x_train)  
  
    # Plot the residual  
    plot_residual(y_train,predicted_y)  
    # View model coefficients  
    view_model(model)  
  
    print "\n Model Performance in Training set\n"  
    model_worth(y_train,predicted_y)  
  
    # Apply the model on dev set  
    predicted_y = model.predict(x_dev)  
    print "\n Model Performance in Dev set\n"  
    model_worth(y_dev,predicted_y)  
  
    #Prepare some polynomial features  
    poly_features = PolynomialFeatures(2)  
    poly_features.fit(x_train)  
    x_train_poly = poly_features.transform(x_train)  
    x_dev_poly = poly_features.transform(x_dev)  
  
    # Build model with polynomial features  
    model_poly = build_model(x_train_poly,y_train)  
    predicted_y = model_poly.predict(x_train_poly)  
    print "\n Model Performance in Training set (Polynomial features)\n"  
    model_worth(y_train,predicted_y)  
  
    # Apply the model on dev set  
    predicted_y = model_poly.predict(x_dev_poly)  
    print "\n Model Performance in Dev set (Polynomial features)\n"  
    model_worth(y_dev,predicted_y)
```

```
# Apply the model on Test set
x_test_poly = poly_features.transform(x_test)
predicted_y = model_poly.predict(x_test_poly)

print "\n Model Performance in Test set (Polynomial features)\n"
model_worth(y_test,predicted_y)

predicted_y = model.predict(x_test)
print "\n Model Performance in Test set (Regular features)\n"
model_worth(y_test,predicted_y)
```

How it works...

Let's start with the main module and follow the code. We will load the predictor `x` and response variable `y` using the `get_data` function:

```
def get_data():
    """
    Return boston dataset
    as x - predictor and
    y - response variable
    """
    data = load_boston()
    x = data['data']
    y = data['target']
    return x,y
```

The function invokes scikit-learn's convenient `load_boston()` function in order to retrieve the Boston house pricing dataset as NumPy arrays.

We will proceed to divide the data into the train and test sets using the `train_test_split` function from the Scikit library. We will reserve 30 percent of our dataset to test:

```
x_train,x_test_all,y_train,y_test_all =
train_test_split(x,y,test_size = 0.3,random_state=9)
```

Out of which, we will extract the dev set in the next line:

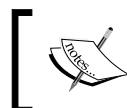
```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)
```

In the next line, we will proceed to build our model using the training dataset by calling the `build_model` method. This model creates an object of a `LinearRegression` type. The `LinearRegression` class encloses SciPy's least squares method:

```
model = LinearRegression(normalize=True,fit_intercept=True)
```

Let's look at the parameters passed when initializing this class.

The `fit_intercept` parameter is set to `True`. This tells the linear regression class to center the data. By centering the data, the mean value of each of our predictors is set to zero. The linear regression methods require the data to be centered by its mean value for a better interpretation of the intercepts. In addition to centering each attribute by its mean, we will also normalize each attribute by its standard deviation. We will achieve this using the `normalize` parameter and setting it to `True`. Refer to the *Chapter 3, Scaling and data standardization* recipes on how to perform normalization by each column. With the `fit_intercept` parameter, we will instruct the algorithm to include an intercept in order to accommodate any constant shift in the response variable. Finally, we will fit the model by invoking the `fit` function with our response variable `y` and predictor `x`.



Refer to the book, *The Elements of Statistical Learning* by Trevor Hastie et al. for more information about linear regression methodologies.

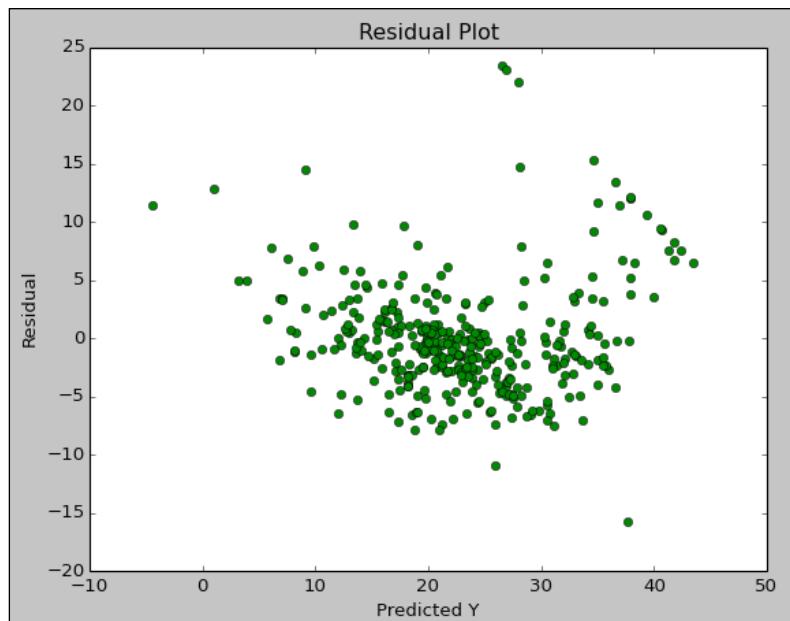


It is good practice to inspect the model that we built so that we can have a better understanding of the model for further improvement or interpretability.

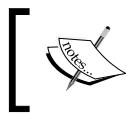
Let's now plot the residuals (the difference between the predicted `y` and actual `y`) and the predicted `y` values as a scatter plot. We will invoke the `plot_residual` method to do this:

```
# Plot the residual
plot_residual(y_train,predicted_y)
```

Let's look at the following graph:



We can validate the regression assumptions in our dataset using this scatter plot. We don't see any pattern and the points are scattered uniformly along zero residual values.



Refer to the book, *Data Mining Methods and Models* by Daniel. T. Larose for more information about using residual plots in order to validate linear regression assumptions.

We will then inspect our model using the `view_model` method. In this method, we will print our intercept and coefficient values. The linear regression object has two attributes, one called `coef_`, which provides us with an array of coefficients, and one called `intercept_`, which gives the intercept value:

```
Model coefficients
=====
Coefficient 1 -0.109
Coefficient 2 0.043
Coefficient 3 0.053
Coefficient 4 2.237
Coefficient 5 -15.879
Coefficient 6 3.883
Coefficient 7 0.001
Coefficient 8 -1.321
Coefficient 9 0.284
Coefficient 10 -0.012
Coefficient 11 -0.904
Coefficient 12 0.009
Coefficient 13 -0.529
Intercept 33.288
```

Let's take `coefficient 6`, which is the number of livable rooms in the house. The coefficient value is interpreted as: for every additional room, the price moves up three times.

Finally, we will look at how good our model is by invoking the `model_worth` function with our predicted response values and actual response values, both from our training and dev sets.

This function prints out the mean squared error value, which is the average square of the difference between the actual and predicted values:

```
Model Performance in Training set
Mean squared error = 23.18
Model Performance in Dev set
Mean squared error = 18.25
```

We have a lower value in our dev set, which is an indication of how good our model is. Let's check whether we can improve our mean squared error. What if we provide more features to our model? Let's create some features from our existing attributes. We will use the `PolynomialFeatures` class from scikit-learn to create second order polynomials:

```
#Prepare some polynomial features
poly_features = PolynomialFeatures(2)
poly_features.fit(x_train)
x_train_poly = poly_features.transform(x_train)
x_dev_poly = poly_features.transform(x_dev)
```

We will pass 2 as a parameter to `PolynomialFeatures` to indicate that we need second order polynomials. 2 is also the default value used if the class is initialized as empty:

```
>>> x_train_poly.shape
(354, 105)
>>> x_train.shape
(354, 13)
```

A quick look at the shape of the new `x` reveals that we now have 105 attributes, compared with 13. Let's build the model using the new polynomial features and check out the model's accuracy:

```
# Build model with polynomial features
model_poly = build_model(x_train_poly,y_train)
predicted_y = model_poly.predict(x_train_poly)
print "\n Model Performance in Training set (Polynomial
features)\n"
model_worth(y_train,predicted_y)

# Apply the model on dev set
predicted_y = model_poly.predict(x_dev_poly)
print "\n Model Performance in Dev set (Polynomial features)\n"
model_worth(y_dev,predicted_y)
```

```
Model Performance in Training set (Polynomial features)
Mean squared error = 5.38

Model Performance in Dev set (Polynomial features)
Mean squared error = 13.20
```

Our model has fitted well with the training dataset. Both in the dev and training sets, our polynomial features performed better than the original features.

Let's finally look at how the model with the polynomial features and the model with the regular features perform with our test set:

```
# Apply the model on Test set
x_test_poly = poly_features.transform(x_test)
predicted_y = model_poly.predict(x_test_poly)

print "\n Model Performance in Test set (Polynomial features)\n"
model_worth(y_test,predicted_y)

predicted_y = model.predict(x_test)
print "\n Model Performance in Test set (Regular features)\n"
model_worth(y_test,predicted_y)
```

Model Performance in Test set (Polynomial features)
Mean squared error = 14.92
Model Performance in Test set (Regular features)
Mean squared error = 21.66

We can see that our polynomial features have fared better than our original set of features using the test dataset.

That is all you need to know about how to do linear regression in Python. We looked at how linear regression works and how we can build models to predict real-valued numbers.

There's more...

Before we move on, we will see one more parameter setting in the `PolyomialFeatures` class called `interaction_only`:

```
poly_features = PolyomialFeatures(interaction_only=True)
```

By setting `interaction_only` to `true`—with `x1` and `x2` attributes—only the `x1*x2` attribute is created. The squares of `x1` and `x2` are not created, assuming that the degree is two.

Our test set results were not as good as our dev set results for both the normal and polynomial features. This is a known problem with linear regression. Linear regression is not well equipped to handle variance. The problem that we are facing is a high variance and low bias. As the model complexity increases, that is, the number of attributes presented to the model increases. The model tends to fit the training data very well—hence a low bias—but starts to give degrading outputs with the test data. There are several techniques available to handle this problem.

Let's look at a method called recursive feature selection. The number of required attributes is passed as a parameter to this method. It recursively filters the features. In the i th run, a linear model is fitted to the data and, based on the coefficients' values, the attributes are filtered; the attributes with lower weights are left out. Thus, the iteration continues with the remaining set of attributes. Finally, when we have the required number of attributes, the iteration stops.

Let's look at a code example:

```
# Load libraries
from sklearn.datasets import load_boston
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from itertools import combinations
from sklearn.feature_selection import RFE

def get_data():
    """
    Return boston dataset
    as x - predictor and
    y - response variable
    """
    data = load_boston()
    x = data['data']
    y = data['target']
    return x,y

def build_model(x,y,no_features):
    """
    Build a linear regression model
    """
    model = LinearRegression(normalize=True,fit_intercept=True)
    rfe_model = RFE(estimator=model,n_features_to_select=no_features)
    rfe_model.fit(x,y)
    return rfe_model

def view_model(model):
    """
    Look at model coeffiecents
    """
    print "\n Model coeffiecents"
    print "=====\\n"
    for i,coef in enumerate(model.coef_):
```

```
        print "\tCoefficient %d  %0.3f"%(i+1,coef)

    print "\n\tIntercept %0.3f"%(model.intercept_)

def model_worth(true_y,predicted_y):
    """
    Evaluate the model
    """
    print "\tMean squared error = %0.2f"%(mean_squared_
error(true_y,predicted_y))
    return mean_squared_error(true_y,predicted_y)

def plot_residual(y,predicted_y):
    """
    Plot residuals
    """
    plt.cla()
    plt.xlabel("Predicted Y")
    plt.ylabel("Residual")
    plt.title("Residual Plot")
    plt.figure(1)
    diff = y - predicted_y
    plt.plot(predicted_y,diff,'go')
    plt.show()

def subset_selection(x,y):
    """
    subset selection method
    """
    # declare variables to track
    # the model and attributes which produces
    # lowest mean square error
    choosen_subset = None
    low_mse = 1e100
    choosen_model = None
    # k value ranges from 1 to the number of
    # attributes in x
    for k in range(1,x.shape[1]+1):
        print "k= %d "%(k)
        # evaluate all attribute combinations
        # of size k+1
        subsets = combinations(range(0,x.shape[1]),k+1)
        for subset in subsets:
            x_subset = x[:,subset]
            model = build_model(x_subset,y)
```

```
predicted_y = model.predict(x_subset)
current_mse = mean_squared_error(y,predicted_y)
if current_mse < low_mse:
    low_mse = current_mse
    choosen_subset = subset
    choosen_model = model

return choosen_model, choosen_subset,low_mse

if __name__ == "__main__":
    x,y = get_data()

    # Divide the data into Train, dev and test
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_size = 0.3,random_state=9)
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_all,test_size=0.3,random_state=9)

    #Prepare some polynomial features
    poly_features = PolynomialFeatures(interaction_only=True)
    poly_features.fit(x_train)
    x_train_poly = poly_features.transform(x_train)
    x_dev_poly = poly_features.transform(x_dev)

    #choosen_model,choosen_subset,low_mse = subset_selection(x_train_poly,y_train)
    choosen_model = build_model(x_train_poly,y_train,20)
    #print choosen_subse
    predicted_y = choosen_model.predict(x_train_poly)
    print "\n Model Performance in Training set (Polynomial features)\n"
    mse = model_worth(y_train,predicted_y)

    # Apply the model on dev set
    predicted_y = choosen_model.predict(x_dev_poly)
    print "\n Model Performance in Dev set (Polynomial features)\n"
    model_worth(y_dev,predicted_y)

    # Apply the model on Test set
    x_test_poly = poly_features.transform(x_test)
    predicted_y = choosen_model.predict(x_test_poly)

    print "\n Model Performance in Test set (Polynomial features)\n"
    model_worth(y_test,predicted_y)
```

This code is very similar to the preceding linear regression code, except for the `build_model` method:

```
def build_model(x,y,no_features):  
    """  
    Build a linear regression model  
    """  
    model = LinearRegression(normalize=True,fit_intercept=True)  
    rfe_model = RFE(estimator=model,n_features_to_select=no_features)  
    rfe_model.fit(x,y)  
    return rfe_model
```

In addition to the predictor `x` and response variable `y`, `build_model` also accepts the number of features to retain `no_features` as a parameter. In this case, we passed a value of 20, asking recursive feature elimination to retain only 20 significant features. As you can see, we first created a linear regression object. This object is passed to the `RFE` class. `RFE` stands for recursive feature elimination, a class provided by scikit-learn to implement recursive feature elimination. Let's now evaluate our model against the training, dev, and test datasets:

```
Model Performance in Training set (Polynomial features)  
Mean squared error = 13.34  
  
Model Performance in Dev set (Polynomial features)  
Mean squared error = 11.51  
  
Model Performance in Test set (Polynomial features)  
Mean squared error = 13.20
```

The mean squared error of the test dataset is 13.20, almost half of what we had earlier. Thus, we are able to use the recursive feature elimination method to perform feature selection effectively and hence improve our model.

See also

- ▶ *Scaling the data recipe in Chapter 3, Data Analysis – Explore and Wrangle*
- ▶ *Standardizing the data recipe in Chapter 3, Data Analysis – Explore and Wrangle*
- ▶ *Preparing data for model building recipe in Chapter 6, Machine Learning I*

Learning regression with L2 shrinkage – ridge

Let's extend the regression technique discussed before to include regularization. While training a linear regression model, some of the coefficients may take very high values, leading to instability in the model. Regularization or shrinkage is a way of controlling the weights of the coefficients such that they don't take very large values. Let's look at the linear regression cost function once again to understand what issues are inherently present with regression, and what we mean by controlling the weights of the coefficients:

$$w = (w_0, w_1, w_2 \dots w_m)$$

$$\sum_{i=1}^n \left(y_i - \sum_{j=0}^m x_j w_j \right)^2$$

Linear regression tries to find the coefficients, $w_0 \dots w_m$, such that it minimizes the preceding equation. There are a few issues with linear regression.

If the dataset contains many correlated predictors, very small changes in the data can lead to an unstable model. Additionally, we will face a problem with interpreting the model results. For example, if we have two variables that are negatively correlated, these variables will have an opposite effect on the response variable. We can manually look at these correlations and remove one of the variables that is responsible and then proceed with the model building. However, it will be helpful if we can handle these scenarios automatically.

We introduced a method called recursive feature elimination in the previous recipe to keep the most informative attributes and discard the rest. However, in this approach, we either keep a variable or don't keep it; our decisions are binary. In this section, we will see a way by which we can control the weights associated with the variables in such a way that the unnecessary variables are penalized heavily and they receive extremely low weights.

We will change the cost function of linear regression to include the coefficients. As you know, the value of the cost function should be at a minimum for the best model. By including the coefficients in the cost function, we can heavily penalize the coefficients that take a very high value. In general, these techniques are known as shrinkage methods, as they try to shrink the value of the coefficients. In this recipe, we will see L2 shrinkage, most commonly called ridge regression. Let's look at the cost function for ridge regression:

$$\sum_{i=1}^n \left(y_i - w_0 - \sum_{j=1}^m x_{ij} w_{ij} \right)^2 + \alpha \sum_{j=1}^m w_j^2$$

As you can see, the sum of the square of the coefficients is added to the cost function. This way, when the optimization routine tries to minimize the preceding function, it has to heavily reduce the value of the coefficients to attain its objective. The alpha parameter decides the amount of shrinkage. Greater the alpha value, greater the shrinkage. The coefficient values are reduced to zero.

With this little math background, let's jump into our recipe to see ridge regression in action.

Getting ready

Once again, we will use the Boston dataset to demonstrate ridge regression. The Boston data has 13 attributes and 506 instances. The target variable is a real number and the median value of the houses is in the thousands. Refer to the following UCI link for more information about the Boston dataset:

<https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names>

We intend to generate the polynomial features of degree two and consider only the interaction effects. At the end of this recipe, we will see how much the coefficients are penalized.

How to do it...

We will start by loading all the necessary libraries. We will follow it up by defining our first function, `get_data()`. In this function, we will read the Boston dataset and return it as predictor `x` and response variable `y`:

```
# Load libraries
from sklearn.datasets import load_boston
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
```

```
from sklearn.preprocessing import PolynomialFeatures

def get_data():
    """
    Return boston dataset
    as x - predictor and
    y - response variable
    """
    data = load_boston()
    x = data['data']
    y = data['target']
    x = x - np.mean(x, axis=0)

    return x, y
```

In our next `build_model` function, we will construct our ridge regression model with the given data. The following two functions, `view_model` and `model_worth`, are used to introspect the model that we built:

```
def build_model(x,y):
    """
    Build a Ridge regression model
    """
    model = Ridge(normalize=True, alpha=0.015)
    model.fit(x,y)
    # Track the scores- Mean squared residual for plot
    return model

def view_model(model):
    """
    Look at model coeffiecents
    """
    print "\n Model coeffiecents"
    print "=====\n"
    for i,coef in enumerate(model.coef_):
        print "\tCoefficient %d %0.3f"%(i+1,coef)

    print "\n\tIntercept %0.3f"%(model.intercept_)

def model_worth(true_y,predicted_y):
    """
    Evaluate the model
    """
    print "\tMean squared error = %0.2f"%(mean_squared_error(true_y,predicted_y))
    return mean_squared_error(true_y,predicted_y)
```

Finally, we will write our `main` function, which is used to invoke all the preceding functions:

```
if __name__ == "__main__":  
  
    x,y = get_data()  
  
    # Divide the data into Train, dev and test  
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_size = 0.3,random_state=9)  
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_all,test_size=0.3,random_state=9)  
  
    #Prepare some polynomial features  
    poly_features = PolynomialFeatures(interaction_only=True)  
    poly_features.fit(x_train)  
    x_train_poly = poly_features.transform(x_train)  
    x_dev_poly = poly_features.transform(x_dev)  
    x_test_poly = poly_features.transform(x_test)  
  
    #chooseen_model,chooseen_subset,low_mse = subset_selection(x_train_poly,y_train)  
    chooseen_model = build_model(x_train_poly,y_train)  
  
    predicted_y = chooseen_model.predict(x_train_poly)  
    print "\n Model Performance in Training set (Polynomial features)\n"  
    mse = model_worth(y_train,predicted_y)  
    view_model(chooseen_model)  
  
    # Apply the model on dev set  
    predicted_y = chooseen_model.predict(x_dev_poly)  
    print "\n Model Performance in Dev set (Polynomial features)\n"  
    model_worth(y_dev,predicted_y)  
  
    # Apply the model on Test set  
    predicted_y = chooseen_model.predict(x_test_poly)  
  
    print "\n Model Performance in Test set (Polynomial features)\n"  
    model_worth(y_test,predicted_y)
```

How it works...

Let's start with the main module and follow the code. We loaded the predictor `x` and response variable `y` using the `get_data` function. This function invokes scikit-learn's convenient `load_boston()` function to retrieve the Boston house pricing dataset as NumPy arrays.

We will proceed to divide the data into train and test sets using the `train_test_split` function from the scikit-learn library. We will reserve 30 percent of our dataset to test. Out of this, we will extract the dev set in the next line.

We will then build the polynomial features:

```
poly_features = PolynomialFeatures(interaction_only=True)
poly_features.fit(x_train)
```

As you can see, we set `interaction_only` to true. By setting `interaction_only` to true—with `x1` and `x2` attributes—only the `x1*x2` attribute is created. The squares of `x1` and `x2` are not created, assuming that the degree is 2. The default degree is two:

```
x_train_poly = poly_features.transform(x_train)
x_dev_poly = poly_features.transform(x_dev)
x_test_poly = poly_features.transform(x_test)
```

Using the `transform` function, we will transform our train, dev, and test datasets to include the polynomial features.

In the next line, we will build our ridge regression model using the training dataset by calling the `build_model` method:

```
model = Ridge(normalize=True, alpha=0.015)
model.fit(x, y)
```

The attributes in the dataset are centered by its mean and standardized by its standard deviation using the `normalize` parameter and setting it to `true`. Alpha controls the amount of shrinkage. Its value is set to `0.015`. We didn't arrive at this number magically, but by running the model several times. Later in this chapter, we will see how to empirically arrive at the right value for this parameter. We will also fit the intercept for this model using the `fit_intercept` parameter. However, by default, the `fit_intercept` parameter is set to `true` and hence we do not specify it explicitly.

Let's now see how the model has performed in the training set. We will call the `model_worth` method to get the mean square error. This method takes the predicted response variable and the actual response variable to return the mean square error:

```
predicted_y = choosen_model.predict(x_train_poly)
print "\n Model Performance in Training set (Polynomial features)\n"
mse = model_worth(y_train,predicted_y)
```

Our output looks as follows:

Model Performance in Training set (Polynomial features)
Mean squared error = 11.49

Before we apply our model to the test set, let's look at the coefficients' weights. We will call a function called `view_model` to view the coefficient's weight:

```
view_model(choosen_model)
```

```
Model coefficients
=====
Coefficient 1  0.000
Coefficient 2  0.066
Coefficient 3  -0.034
Coefficient 4  0.204
Coefficient 5  3.436
Coefficient 6  5.104
Coefficient 7  7.879
Coefficient 8  0.050
Coefficient 9  -0.740
Coefficient 10 0.275
Coefficient 11 0.006
Coefficient 12 -0.061
Coefficient 13 0.001
Coefficient 14 0.206
Coefficient 15 0.265
Coefficient 16 0.003
Coefficient 17 2.896
Coefficient 18 -0.088
Coefficient 19 -0.010
Coefficient 20 0.001
Coefficient 21 -0.187
Coefficient 22 -0.000
Coefficient 23 0.000
Coefficient 24 0.003
Coefficient 25 -0.000
Coefficient 26 0.004
Coefficient 27 -0.003
Coefficient 28 -0.069
Coefficient 29 -0.056
Coefficient 30 0.007
Coefficient 31 0.000
```

We have not shown all the coefficients. There are a total of 92. However, looking at some of them, the shrinkage effect should be visible. For instance, Coefficient 1 is almost 0 (remember that it is a very small value and we have shown only the first three decimal places here).

Let's proceed to see how our model has performed in the dev set:

```
predicted_y = choosen_model.predict(x_dev_poly)
print "\n Model Performance in Dev set (Polynomial features)\n"
model_worth(y_dev,predicted_y)
```

```
Model Performance in Dev set (Polynomial features)
Mean squared error = 10.47
```

Not bad, we have reached a mean square error lower than our training error. Finally, let's look at our model performance on the test set:

```
Model Performance in Test set (Polynomial features)  
Mean squared error = 12.65
```

Compared with our linear regression model in the previous recipe, we performed better on our test set.

There's more...

We mentioned earlier that linear regression models are very sensitive to even small changes in the dataset. Let's see a small example that will demonstrate this:

```
# Load libraries  
from sklearn.datasets import load_boston  
from sklearn.cross_validation import train_test_split  
from sklearn.linear_model import Ridge  
from sklearn.metrics import mean_squared_error  
from sklearn.preprocessing import PolynomialFeatures  
  
def get_data():  
    """  
        Return boston dataset  
        as x - predictor and  
        y - response variable  
    """  
    data = load_boston()  
    x = data['data']  
    y = data['target']  
    x = x - np.mean(x, axis=0)  
  
    return x, y
```

In this code, we will fit both the linear regression and ridge regression models on the original data using the `build_model` function:

```
lin_model, ridg_model = build_model(x,y)
```

We will introduce a small noise in our original data, as follows:

```
# Add some noise to the dataset  
noise = np.random.normal(0,1,(x.shape))  
x = x + noise
```

Once again, we will fit the models on the noisy dataset. Finally, we will compare the coefficients' weights:

```
Linear Regression Model Before Noise
Model coefficients
=====
Coefficient 1 -0.107
Coefficient 2 0.046
Coefficient 3 0.021
Coefficient 4 2.689
Coefficient 5 -17.796
Coefficient 6 3.805
Coefficient 7 0.001
Coefficient 8 -1.476
Coefficient 9 0.306
Coefficient 10 -0.012
Coefficient 11 -0.953
Coefficient 12 0.009
Coefficient 13 -0.525

Intercept 36.491

Linear Regression Model after Noise
Model coefficients
=====
Coefficient 1 -0.095
Coefficient 2 0.053
Coefficient 3 -0.045
Coefficient 4 0.209
Coefficient 5 0.216
Coefficient 6 0.841
Coefficient 7 0.029
Coefficient 8 -0.659
Coefficient 9 0.349
Coefficient 10 -0.017
Coefficient 11 -0.713
Coefficient 12 0.009
Coefficient 13 -0.751

Intercept 41.003
```

After adding a small noise, when we try to fit a model using linear regression, the weights assigned are very different to the the weights assigned by the previous model. Now, let's see how ridge regression performs:

```
Ridge Model Before Noise
Model coefficients
=====
Coefficient 1 -0.004
Coefficient 2 0.001
Coefficient 3 -0.006
Coefficient 4 0.063
Coefficient 5 -0.317
Coefficient 6 0.088
Coefficient 7 -0.001
Coefficient 8 0.010
Coefficient 9 -0.004
Coefficient 10 -0.000
Coefficient 11 -0.021
Coefficient 12 0.000
Coefficient 13 -0.009
Intercept 22.775

Ridge Model after Noise
Model coefficients
=====
Coefficient 1 -0.004
Coefficient 2 0.001
Coefficient 3 -0.006
Coefficient 4 0.001
Coefficient 5 -0.000
Coefficient 6 0.028
Coefficient 7 -0.001
Coefficient 8 0.008
Coefficient 9 -0.004
Coefficient 10 -0.000
Coefficient 11 -0.017
Coefficient 12 0.000
Coefficient 13 -0.009
Intercept 22.917
```

The weights have not varied starkly between the first and second model. Hopefully, this demonstrates the stability of ridge regression under noisy data conditions.

It is always tricky to choose the appropriate alpha value. A brute force approach is to run it through multiple values and trace the path of the coefficients. From the path, choose the alpha value where the weights don't vary dramatically. We will plot the coefficients' weights using the `coeff_path` function.

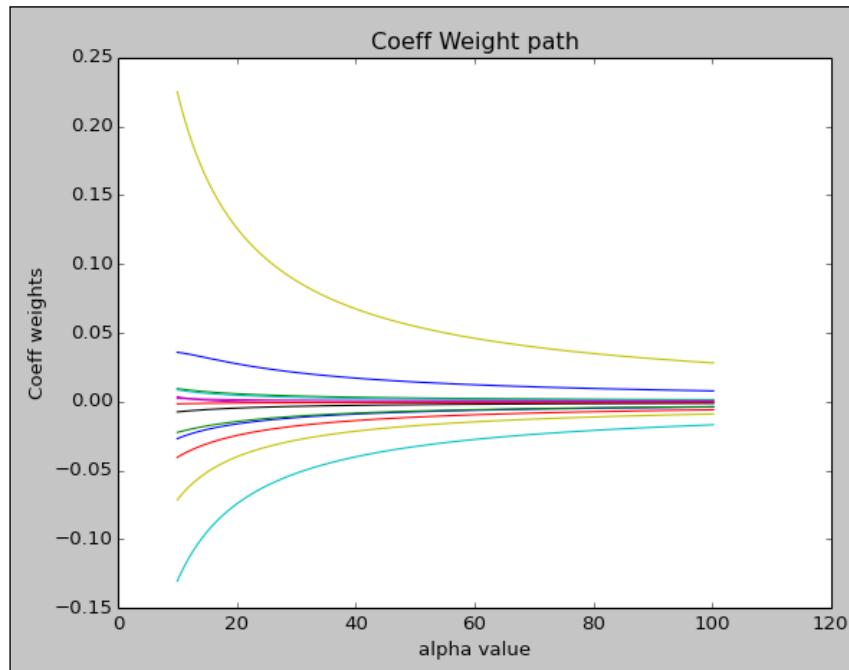
Let's look at the `coeff_path` function. It first generates a list of the alpha values:

```
alpha_range = np.linspace(10, 100.2, 300)
```

In this case, we generated 300 uniformly spaced numbers between 10 and 100. For each of these alpha values, we will build a model and save its coefficients:

```
for alpha in alpha_range:  
    model = Ridge(normalize=True, alpha=alpha)  
    model.fit(x, y)  
    coeffs.append(model.coef_)
```

Finally, we will plot these coefficient weights against the alpha value:



As you can see, the values stabilize around the alpha value of 100. You can further zoom into a range close to 100 and look for an ideal value.

See also

- ▶ *Predicting real valued numbers using regression recipe in Chapter 7, Machine Learning II*
- ▶ *Scaling the data recipe in Chapter 3, Data Analysis – Explore and Wrangle*
- ▶ *Standardizing the data recipe in Chapter 3, Data Analysis – Explore and Wrangle*
- ▶ *Preparing data for model building recipe in Chapter 6, Machine Learning I*

Learning regression with L1 shrinkage – LASSO

Least absolute shrinkage and selection operator (LASSO) is another shrinkage method popularly used with regression problems. LASSO leads to sparse solutions compared with ridge. A solution is called sparse if most of the coefficients are reduced to zero. In LASSO, a lot of the coefficients are made zero. In the case of correlated variables, LASSO selects only one of them, whereas ridge assigns equal weights to the coefficients of both the variables. This attribute of LASSO can hence be leveraged for variable selection. In this recipe, let's see how we can leverage LASSO for variable selection.

Let's look at the cost function of LASSO regression. If you followed through the previous two recipes, you can quickly identify the difference:

$$\sum_{i=1}^n \left(y_i - w_0 - \sum_{j=1}^m x_{ij} w_{ij} \right)^2 + \alpha \sum_{j=1}^m |w_j|$$

The coefficients are penalized by the sum of the absolute value of the coefficients. Once again, the alpha controls the level of penalization. Let's try to understand the intuition behind why L1 shrinkage leads to a sparse solution.

We can rewrite the preceding equation as an unconstrained cost function and a constraint, as follows:

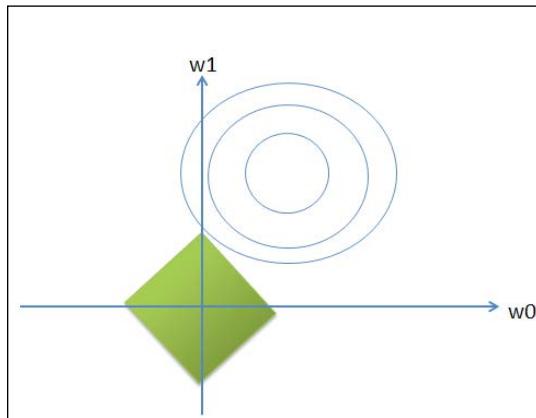
Minimize:

$$\sum_{i=1}^n \left(y_i - w_0 - \sum_{j=1}^m x_{ij} w_{ij} \right)^2$$

Subject to the constraint:

$$\sum_{j=1}^m |w_j| \leq \eta$$

With this equation in mind, let's plot the cost function values in the coefficient space for two coefficients, w_0 and w_1 :



The blue lines represent the contours of the cost function (without constraint) values for the different values of w_0 and w_1 . The green region represents the constraint shape dictated by the eta value. The optimized value where both the regions meet is when w_0 is set to 0. We depicted a two-dimensional space where our solution is made sparse with w_0 set to 0. In a multidimensional space, we will have a rhomboid in the green region, and LASSO will give a sparse solution by reducing many of the coefficients to zero.

Getting ready

Once again, we will use the Boston dataset to demonstrate LASSO regression. The Boston data has 13 attributes and 506 instances. The target variable is a real number and the median value of the houses is in the thousands.

Refer to the following UCI link for more information on the Boston dataset:

<https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names>.

We will see how we can use LASSO for variable selection.

How to do it...

We will start by loading all the necessary libraries. We will follow it up by defining our first function, `get_data()`. In this function, we will read the Boston dataset and return it as a predictor `x` and response variable `y`:

```
# Load libraries
from sklearn.datasets import load_boston
```

```
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import Lasso, LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
import numpy as np

def get_data():
    """
    Return boston dataset
    as x - predictor and
    y - response variable
    """
    data = load_boston()
    x = data['data']
    y = data['target']
    return x,y
```

In our next `build_model` function, we will construct our LASSO regression model with the given data. The following two functions, `view_model` and `model_worth`, are used to introspect the model that we built:

```
def build_models(x,y):
    """
    Build a Lasso regression model
    """
    # Alpha values uniformly
    # spaced between 0.01 and 0.02
    alpha_range = np.linspace(0,0.5,200)
    model = Lasso(normalize=True)
    coefficients = []
    # Fit a model for each alpha value
    for alpha in alpha_range:
        model.set_params(alpha=alpha)
        model.fit(x,y)
        # Track the coefficients for plot
        coefficients.append(model.coef_)
    # Plot coefficients weight decay vs alpha value
    # Plot model RMSE vs alpha value
    coeff_path(alpha_range,coefficients)
    # View coefficient value
    #view_model(model)

def view_model(model):
    """
```

```
Look at model coefficients
"""
print "\n Model coefficients"
print "=====\\n"
for i,coef in enumerate(model.coef_):
    print "\\tCoefficient %d %0.3f"%(i+1,coef)

print "\\n\\tIntercept %0.3f"%(model.intercept_)

def model_worth(true_y,predicted_y):
"""
Evaluate the model
"""
    print "\\t Mean squared error = %0.2f\\n"%(mean_squared_error(true_y,predicted_y))
```

We will define two functions, `coeff_path` and `get_coeff`, to inspect our model coefficients. The `coeff_path` function is invoked from the `build_model` function to plot the weights of the coefficients for different alpha values. The `get_coeff` function is invoked from the main function:

```
def coeff_path(alpha_range,coefficients):
"""
Plot residuals
"""
plt.close('all')
plt.cla()

plt.figure(1)
plt.xlabel("Alpha Values")
plt.ylabel("Coefficient Weight")
plt.title("Coefficient weights for different alpha values")
plt.plot(alpha_range,coefficients)
plt.axis('tight')

plt.show()

def get_coeff(x,y,alpha):
    model = Lasso(normalize=True,alpha=alpha)
    model.fit(x,y)
    coefs = model.coef_
    indices = [i for i,coef in enumerate(coefs) if abs(coef) > 0.0]
    return indices
```

Finally, we will write our `main` function, which is used to invoke all the preceding functions:

```
if __name__ == "__main__":  
  
    x,y = get_data()  
    # Build multiple models for different alpha values  
    # and plot them  
    build_models(x,y)  
    print "\nPredicting using all the variables"  
    full_model = LinearRegression(normalize=True)  
    full_model.fit(x,y)  
    predicted_y = full_model.predict(x)  
    model_worth(y,predicted_y)  
  
    print "\nModels at different alpha values\n"  
    alpa_values = [0.22,0.08,0.01]  
    for alpha in alpa_values:  
  
        indices = get_coeff(x,y,alpha)  
        print "\t alpah =%0.2f Number of variables selected = %d  
"%(alpha,len(indices))  
        print "\t attributes include ", indices  
        x_new = x[:,indices]  
        model = LinearRegression(normalize=True)  
        model.fit(x_new,y)  
        predicted_y = model.predict(x_new)  
        model_worth(y,predicted_y)
```

How it works...

Let's start with the main module and follow the code. We will load the predictor `x` and response variable `y` using the `get_data` function. The function invokes scikit-learn's convenient `load_boston()` function to retrieve the Boston house pricing dataset as NumPy arrays.

We will proceed by calling `build_models`. In `build_models`, we will construct multiple models for the different values of `alpha`:

```
alpha_range = np.linspace(0,0.5,200)  
model = Lasso(normalize=True)  
coeffiecents = []  
# Fit a model for each alpha value  
for alpha in alpha_range:  
    model.set_params(alpha=alpha)  
    model.fit(x,y)  
    # Track the coeffiecents for plot  
    coeffiecents.append(model.coef_)
```

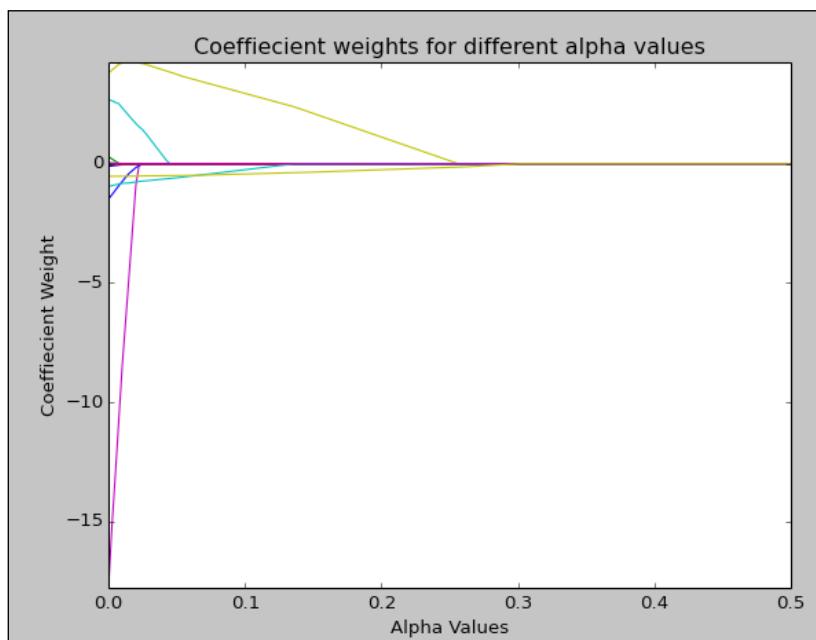
As you can see, in the for loop, we also store the coefficient values for different values of alpha in a list.

Let's plot the coefficient values for different alpha values by calling the `coeff_path` function:

```
plt.close('all')
plt.cla()

plt.figure(1)
plt.xlabel("Alpha Values")
plt.ylabel("Coefficient Weight")
plt.title("Coefficient weights for different alpha values")
plt.plot(alpha_range, coefficients)
plt.axis('tight')
plt.show()
```

In the x axis, you can see that we have the alpha values, and in the y axis, we will plot all the coefficients for a given alpha value. Let's see the output plot:



The different colored lines represent different coefficient values. As you can see, as the value of alpha increases, the coefficient weights merge towards zero. From this plot, we can select the value of alpha.

For our reference, let's fit a simple linear regression model:

```
print "\nPredicting using all the variables"
full_model = LinearRegression(normalize=True)
full_model.fit(x,y)
predicted_y = full_model.predict(x)
model_worth(y,predicted_y)
```

Let's look at the mean square error when we try to predict using our newly built model:

Predicting using all the variables
Mean squared error = 21.90

Let's proceed to select the coefficients based on LASSO:

```
print "\nModels at different alpha values\n"
alpa_values = [0.22,0.08,0.01]
for alpha in alpa_values:
    indices = get_coeff(x,y,alpha)
```

Based on our preceding graph, we selected 0.22, 0.08, and 0.01 as the alpha values. In the loop, we will call the `get_coeff` method. This method fits a LASSO model with the given alpha values and returns only the non-zero coefficients' indices:

```
model = Lasso(normalize=True,alpha=alpha)
model.fit(x,y)
coefs = model.coef_

indices = [i for i,coef in enumerate(coefs) if abs(coef) > 0.0]
```

Essentially, we are selecting only those attributes that have a non-zero coefficient value—feature selection. Let's get back to our `for` loop where we will fit a linear regression model with the reduced coefficients:

```
print "\t alpah =%0.2f Number of variables selected = %d
"%(alpha,len(indices))
print "\t attributes include ", indices
x_new = x[:,indices]
model = LinearRegression(normalize=True)
model.fit(x_new,y)
predicted_y = model.predict(x_new)
model_worth(y,predicted_y)
```

What we want to know is how good our models would be if we predicted them with the reduced set of attributes, compared with the model that we built initially using the whole dataset:

```
Models at different alpha values

alpha =0.22 Number of variables selected = 2
attributes include [5, 12]
Mean squared error = 30.51

alpha =0.08 Number of variables selected = 3
attributes include [5, 10, 12]
Mean squared error = 27.13

alpha =0.01 Number of variables selected = 9
attributes include [0, 1, 3, 4, 5, 7, 10, 11, 12]
Mean squared error = 22.89
```

Look at the first pass where our alpha value is 0.22. There are only two coefficients with non-zero values, 5 and 12. The mean squared error is 30.51, which is only 9 more than the model fitted with all the variables.

Similarly, for the alpha value of 0.08, there are three non-zero coefficients. We can see some improvement in the mean squared error. Finally, with 0.01 alpha value, 9 out of 13 attributes are selected and the mean square error is very close to the model built with all the attributes.

As you can see, we didn't fit the model with all the attributes. We are able to choose a subset of the attributes automatically using LASSO. Thus, we have seen how LASSO can be used for variable selection.

There's more...

By keeping only the most important variables, LASSO avoids overfitting. However, as you can see, the mean squared error values are not that good. We can see that there is a loss in the predictive power because of LASSO.

As said before, in the case of the correlated variables, LASSO selects only one of them, whereas ridge assigns equal weights to the coefficients of both the variables. Hence, ridge has a higher predictive power compared with LASSO. However, LASSO can do variable selection, which Ridge is not capable of performing.



Refer to the book, *Statistical learning with sparsity: The Lasso and generalization* by Trevor Hastie et al. for more information about the LASSO and ridge methodologies.

See also

- ▶ *Scaling the data recipe in Chapter 3, Data Analysis – Explore and Wrangle*
- ▶ *Standardizing the data recipe in Chapter 3, Data Analysis – Explore and Wrangle*
- ▶ *Preparing data for model building recipe in Chapter 6, Machine Learning I*
- ▶ *Regression with L2 Shrinkage – Ridge recipe in Chapter 7, Machine Learning II*

Using cross-validation iterators with L1 and L2 shrinkage

In the previous chapter, we saw methods to divide the data into train and test sets. In the subsequent recipes, we again performed a split on the test dataset to arrive at a dev dataset. The idea was to keep the test set away from the model building cycle. However, as we need to improve our model continuously, we used the dev set to test the model accuracy in each iteration. Though it's a good approach, this method is difficult to implement if we don't have a large dataset. We want to provide as much data as possible to train our model but still need to hold some of the data for the evaluation and final testing. In many real-world scenarios, it is very rare to get a very large dataset.

In this recipe, we will see a method called cross-validation to help us address this issue. This approach is typically called k-fold cross-validation. The training set is divided into k-folds. The model is trained on K-1 (K minus 1) folds and the left out fold is used to test. This way, we don't need a separate dev dataset.

Let's see some of the iterators provided by the scikit-learn library to perform the k-fold cross-validation effectively. Equipped with the knowledge of cross-validation, we will further see how we can leverage cross-validation for the selection of the alpha values in shrinkage methods.

Getting ready

We will use the Iris dataset to demonstrate the various cross-validation iterator concepts. We will return to our Boston housing dataset to demonstrate how cross-validation can be used successfully to find the ideal alpha values in shrinkage methods.

How to do it...

Let's look at how to use the cross validation iterator:

```
from sklearn.datasets import load_iris
from sklearn.cross_validation import KFold, StratifiedKFold
```

```
def get_data():
    data = load_iris()
    x = data['data']
    y = data['target']
    return x,y

def class_distribution(y):
    class_dist = {}
    total = 0
    for entry in y:
        try:
            class_dist[entry]+=1
        except KeyError:
            class_dist[entry]=1
        total+=1

    for k,v in class_dist.items():
        print "\tclass %d percentage =%0.2f"%(k,v/(1.0*total))

if __name__ == "__main__":
    x,y = get_data()
    # K Fold
    # 3 folds
    kfolds = KFold(n=y.shape[0],n_folds=3)
    fold_count =1
    print
    for train,test in kfolds:
        print "Fold %d x train shape"%(fold_count),x[train].shape,
              " x test shape",x[test].shape
        fold_count==1
    print
    #Stratified KFold
    skfolds = StratifiedKFold(y,n_folds=3)
    fold_count =1
    for train,test in skfolds:
        print "\nFold %d x train shape"%(fold_count),x[train].shape,
              " x test shape",x[test].shape
        y_train = y[train]
        y_test = y[test]
        print "Train Class Distribution"
        class_distribution(y_train)
        print "Test Class Distribution"
        class_distribution(y_test)

    fold_count+=1

print
```

In our main function, we will call the `get_data` function to load the Iris dataset. We will then proceed to demonstrate a simple k-fold and stratified k-folds.

With the knowledge of k-fold cross-validation, let's write a recipe to leverage this newly found knowledge in enhancing ridge regression:

```
# Load libraries
from sklearn.datasets import load_boston
from sklearn.cross_validation import KFold,train_test_split
from sklearn.linear_model import Ridge
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import PolynomialFeatures
import numpy as np

def get_data():
    """
    Return boston dataset
    as x - predictor and
    y - response variable
    """
    data = load_boston()
    x    = data['data']
    y    = data['target']
    return x,y
```

We will start by loading all the necessary libraries. We will follow it up by defining our first function, `get_data()`. In this function, we will read the Boston dataset and return it as a predictor `x` and response variable `y`.

In our next `build_model` function, we will construct our ridge regression model with the given data. We will leverage the k-fold cross-validation.

The following two functions, `view_model` and `model_worth`, are used to introspect the model that we built.

Finally, we will write the `display_param_results` function to view the model errors in each fold:

```
def build_model(x,y):
    """
    Build a Ridge regression model
    """
    kfold = KFold(y.shape[0],5)
    model = Ridge(normalize=True)

    alpha_range = np.linspace(0.0015,0.0017,30)
```

```
grid_param = {"alpha":alpha_range}
grid = GridSearchCV(estimator=model,param_grid=grid_
param,cv=kfold,scoring='mean_squared_error')
grid.fit(x,y)
display_param_results(grid.grid_scores_)
print grid.best_params_
# Track the scores- Mean squared residual for plot
return grid.best_estimator_

def view_model(model):
"""
Look at model coeffiecents
"""
#print "\n Estimated Alpha = %0.3f"%model.alpha_
print "\n Model coeffiecents"
print "=====\\n"
for i,coef in enumerate(model.coef_):
    print "\\tCoefficient %d %0.3f"%(i+1,coef)

print "\\n\\tIntercept %0.3f"%(model.intercept_)

def model_worth(true_y,predicted_y):
"""
Evaluate the model
"""
print "\\tMean squared error = %0.2f"%(mean_squared_
error(true_y,predicted_y))
return mean_squared_error(true_y,predicted_y)

def display_param_results(param_results):
    fold = 1
    for param_result in param_results:
        print "Fold %d Mean squared error %0.2f"%(fold,abs(param_
result[1])),param_result[0]
    fold+=1
```

Finally, we will write our `main` function, which is used to invoke all the preceding functions:

```
if __name__ == "__main__":
    x,y = get_data()

    # Divide the data into Train and test
    x_train,x_test,y_train,y_test = train_test_split(x,y,test_size =
0.3,random_state=9)

    #Prepare some polynomial features
```

```
poly_features = PolynomialFeatures(interaction_only=True)
poly_features.fit(x_train)
x_train_poly = poly_features.transform(x_train)
x_test_poly = poly_features.transform(x_test)

chooseen_model = build_model(x_train_poly,y_train)
predicted_y = chooseen_model.predict(x_train_poly)
model_worth(y_train,predicted_y)

view_model(chooseen_model)

predicted_y = chooseen_model.predict(x_test_poly)
model_worth(y_test,predicted_y)
```

How it works...

Let's start with our main method. We will start with the `KFold` class. This iterator class is instantiated with the number of instances in our dataset and the number of folds that we require:

```
kfolds = KFold(n=y.shape[0],n_folds=3)
```

Now, we can iterate through the folds, as follows:

```
fold_count =1
print
for train,test in kfolds:
    print "Fold %d x train shape"%(fold_count),x[train].shape,\n    " x test shape",x[test].shape
    fold_count==1
```

Let's see the print statement output:

```
Fold 1 x train shape (100, 4) x test shape (50, 4)
Fold 1 x train shape (100, 4) x test shape (50, 4)
Fold 1 x train shape (100, 4) x test shape (50, 4)
```

We can see that the data is split into three parts, each with 100 instances to train and 50 instances to test.

We will move on next to `StratifiedKFold`. Recall our discussion on having a uniform class distribution in the train and test split from the previous chapter. `StratifiedKFold` achieves a uniform class distribution across the three folds.

It is invoked as follows:

```
skfolds = StratifiedKFold(y,n_folds=3)
```

As it needs to know the distribution of the class label in the dataset, this iterator object takes the response variable `y` as one of its parameters. The other parameter is the number of folds requested.

Let's print the shape of our train and test sets in these three folds, along with their class distribution. We will use the `class_distribution` function to print the distribution of the classes in each of the folds:

```
fold_count =1
for train,test in skfolds:
    print "\nFold %d x train shape"%(fold_count),x[train].shape,\n      " x test shape",x[test].shape
    y_train = y[train]
    y_test = y[test]
    print "Train Class Distribution"
    class_distribution(y_train)
    print "Test Class Distribution"
    class_distribution(y_test)

    fold_count+=1
```

```
Fold 1 x train shape (99, 4)  x test shape (51, 4)
Train Class Distribution
    class 0 percentage =0.33
    class 1 percentage =0.33
    class 2 percentage =0.33
Test Class Distribution
    class 0 percentage =0.33
    class 1 percentage =0.33
    class 2 percentage =0.33

Fold 2 x train shape (99, 4)  x test shape (51, 4)
Train Class Distribution
    class 0 percentage =0.33
    class 1 percentage =0.33
    class 2 percentage =0.33
Test Class Distribution
    class 0 percentage =0.33
    class 1 percentage =0.33
    class 2 percentage =0.33

Fold 3 x train shape (102, 4)  x test shape (48, 4)
Train Class Distribution
    class 0 percentage =0.33
    class 1 percentage =0.33
    class 2 percentage =0.33
Test Class Distribution
    class 0 percentage =0.33
    class 1 percentage =0.33
    class 2 percentage =0.33
```

You can see that the classes are distributed uniformly.

Let's assume that you build a five-fold dataset, you fit five different models, and you have five different accuracy scores. You can now take the mean of these scores to evaluate how good your model has turned out to be. If you are not satisfied, you can go ahead and start rebuilding your model with a different set of parameters and again run it on the five-fold data and see the mean accuracy score. This way, you can continuously improve the model by finding the right parameter values only using the training dataset.

Armed with this knowledge, let's revisit our old ridge regression problem.

Let's start with the `main` module and follow the code. We will load the predictor `x` and response variable `y` using the `get_data` function. This function invokes scikit-learn's convenient `load_boston()` function to retrieve the Boston house pricing dataset as NumPy arrays.

We will proceed to divide the data into train and test sets using the `train_test_split` function from the scikit-learn library. We will reserve 30 percent of our dataset to test.

We will then build the polynomial features:

```
poly_features = PolynomialFeatures(interaction_only=True)
poly_features.fit(x_train)
```

As you can see, we set `interaction_only` to `true`. By setting `interaction_only` to `true`—with `x1` and `x2` attributes—only the `x1*x2` attribute is created. The squares of `x1` and `x2` are not created, assuming that the degree is two. The default degree is two:

```
x_train_poly = poly_features.transform(x_train)
x_test_poly = poly_features.transform(x_test)
```

Using the `transform` function, we will transform our train and test dataset to include the polynomial features. Let's call the `build_model` function. The first thing that we notice in the `build_model` function is the `k`-fold declaration. We will apply our knowledge of cross-validation here and create a five-fold dataset:

```
kfold = KFold(y.shape[0], 5)
```

We will then create our ridge object:

```
model = Ridge(normalize=True)
```

Let's now see how we can leverage our `k`-folds to figure out the ideal alpha value for our ridge regression. In the next line, we will create an object out of `GridSearchCV`:

```
grid = GridSearchCV(estimator=model, param_grid=grid_
param, cv=kfold, scoring='mean_squared_error')
```

GridSearchCV is a convenient function from scikit-learn that helps us train our models with a range of parameters. In this case, we want to find the ideal alpha value, and hence, would like to train our models with different alpha values. Let's look at the parameters passed to GridSearchCV:

Estimator: This is the type of model that should be run with the given parameter and data. In our case, we want to run ridge regression. Hence, we will create a ridge object and pass it to GridSearchCV.

Param-grid: This is a dictionary of parameters that we want to evaluate our model on. Let's work this through in detail. We will first declare the range of alpha values that we want to build our model on:

```
alpha_range = np.linspace(0.0015, 0.0017, 30)
```

This gives us a NumPy array of 30 uniformly spaced elements starting from 0.0015 and ending at 0.0017. We want to build a model for each of these values. We will create a dictionary object called `grid_param` and make an entry under the `alpha` key with the generated NumPy array of alpha values:

```
grid_param = {"alpha":alpha_range}
```

We will pass this dictionary as a parameter to GridSearchCV. Look at the entry, `param_grid=grid_param`.

Cv: This defines the kind of cross-validation that we are interested in. We will pass the k-fold (five-fold) iterator that we created before as the `cv` parameter.

Finally, we need to define a scoring function. In our case, we are interested in finding out the squared error. This is the metric with which we will evaluate our model.

So, internal GridSearchCV will build five models for each of our parameter values and return the mean score when tested in the left out folds. In our case, we have five folds of test data, so the average of the score values across these five folds of test data is returned.

With this explained, we will then fit our model, that is, start our grid search activity.

Finally, we want to see the output at the various parameter settings. We will use the `display_param_results` function to display the average mean squared error across the different folds:

```

Fold 1 Mean squared error 14.24 {'alpha': 0.0015}
Fold 2 Mean squared error 14.24 {'alpha': 0.0015068965517241379}
Fold 3 Mean squared error 14.24 {'alpha': 0.0015137931034482758}
Fold 4 Mean squared error 14.24 {'alpha': 0.0015206896551724139}
Fold 5 Mean squared error 14.24 {'alpha': 0.0015275862068965518}
Fold 6 Mean squared error 14.24 {'alpha': 0.0015344827586206897}
Fold 7 Mean squared error 14.24 {'alpha': 0.0015413793103448276}
Fold 8 Mean squared error 14.24 {'alpha': 0.0015482758620689655}
Fold 9 Mean squared error 14.24 {'alpha': 0.0015551724137931034}
Fold 10 Mean squared error 14.24 {'alpha': 0.0015620689655172415}
Fold 11 Mean squared error 14.24 {'alpha': 0.0015689655172413794}
Fold 12 Mean squared error 14.24 {'alpha': 0.0015758620689655172}
Fold 13 Mean squared error 14.24 {'alpha': 0.0015827586206896551}
Fold 14 Mean squared error 14.24 {'alpha': 0.001589655172413793}
Fold 15 Mean squared error 14.24 {'alpha': 0.0015965517241379309}
Fold 16 Mean squared error 14.24 {'alpha': 0.001603448275862069}
Fold 17 Mean squared error 14.24 {'alpha': 0.0016103448275862069}
Fold 18 Mean squared error 14.24 {'alpha': 0.0016172413793103448}
Fold 19 Mean squared error 14.24 {'alpha': 0.0016241379310344827}
Fold 20 Mean squared error 14.25 {'alpha': 0.0016310344827586206}
Fold 21 Mean squared error 14.25 {'alpha': 0.0016379310344827587}
Fold 22 Mean squared error 14.25 {'alpha': 0.0016448275862068966}
Fold 23 Mean squared error 14.25 {'alpha': 0.0016517241379310345}
Fold 24 Mean squared error 14.25 {'alpha': 0.0016586206896551724}
Fold 25 Mean squared error 14.25 {'alpha': 0.0016655172413793102}
Fold 26 Mean squared error 14.25 {'alpha': 0.0016724137931034481}
Fold 27 Mean squared error 14.25 {'alpha': 0.001679310344827586}
Fold 28 Mean squared error 14.25 {'alpha': 0.0016862068965517241}
Fold 29 Mean squared error 14.25 {'alpha': 0.001693103448275862}
Fold 30 Mean squared error 14.25 {'alpha': 0.0016999999999999999}

```

Each line in the output displays the parameter alpha value and average mean squared error from the test folds. We can see that as we move deep into the 0.0016 range, the mean square error is increasing. Hence, we decide to stop at 0.0015. We can query the grid object to get the best parameter and estimator:

```

print grid.best_params_
return grid.best_estimator_

```

This was not the first set of alpha values that we tested it with. Our initial alpha values were as follows:

```
alpha_range = np.linspace(0.01, 1.0, 30)
```

The following was our output:

```
Fold 1 Mean squared error 16.24 {'alpha': 0.01}
Fold 2 Mean squared error 19.80 {'alpha': 0.044137931034482762}
Fold 3 Mean squared error 21.25 {'alpha': 0.078275862068965515}
Fold 4 Mean squared error 22.08 {'alpha': 0.11241379310344828}
Fold 5 Mean squared error 22.65 {'alpha': 0.14655172413793105}
Fold 6 Mean squared error 23.07 {'alpha': 0.18068965517241381}
Fold 7 Mean squared error 23.41 {'alpha': 0.21482758620689657}
Fold 8 Mean squared error 23.70 {'alpha': 0.24896551724137933}
Fold 9 Mean squared error 23.94 {'alpha': 0.28310344827586209}
Fold 10 Mean squared error 24.15 {'alpha': 0.31724137931034485}
Fold 11 Mean squared error 24.35 {'alpha': 0.35137931034482761}
Fold 12 Mean squared error 24.53 {'alpha': 0.38551724137931037}
Fold 13 Mean squared error 24.69 {'alpha': 0.41965517241379313}
Fold 14 Mean squared error 24.84 {'alpha': 0.45379310344827589}
Fold 15 Mean squared error 24.99 {'alpha': 0.48793103448275865}
Fold 16 Mean squared error 25.12 {'alpha': 0.52206896551724147}
Fold 17 Mean squared error 25.26 {'alpha': 0.55620689655172417}
Fold 18 Mean squared error 25.38 {'alpha': 0.59034482758620688}
Fold 19 Mean squared error 25.50 {'alpha': 0.62448275862068969}
Fold 20 Mean squared error 25.62 {'alpha': 0.65862068965517251}
Fold 21 Mean squared error 25.73 {'alpha': 0.69275862068965521}
Fold 22 Mean squared error 25.84 {'alpha': 0.72689655172413792}
Fold 23 Mean squared error 25.95 {'alpha': 0.76103448275862073}
Fold 24 Mean squared error 26.06 {'alpha': 0.79517241379310355}
Fold 25 Mean squared error 26.16 {'alpha': 0.82931034482758625}
Fold 26 Mean squared error 26.26 {'alpha': 0.86344827586206896}
Fold 27 Mean squared error 26.36 {'alpha': 0.89758620689655177}
Fold 28 Mean squared error 26.46 {'alpha': 0.93172413793103459}
Fold 29 Mean squared error 26.56 {'alpha': 0.9658620689655173}
Fold 30 Mean squared error 26.65 {'alpha': 1.0}
```

When our alpha values were above 0.01, the mean squared error was shooting up. Hence, we again gave a new range:

```
alpha_range = np.linspace(0.001, 0.1, 30)
```

Our output was as follows:

```
Fold 1 Mean squared error 14.33 {'alpha': 0.001}
Fold 2 Mean squared error 14.92 {'alpha': 0.004413793103448276}
Fold 3 Mean squared error 15.78 {'alpha': 0.0078275862068965529}
Fold 4 Mean squared error 16.48 {'alpha': 0.011241379310344829}
Fold 5 Mean squared error 17.05 {'alpha': 0.014655172413793105}
Fold 6 Mean squared error 17.54 {'alpha': 0.018068965517241381}
Fold 7 Mean squared error 17.95 {'alpha': 0.021482758620689657}
Fold 8 Mean squared error 18.32 {'alpha': 0.024896551724137933}
Fold 9 Mean squared error 18.65 {'alpha': 0.028310344827586209}
Fold 10 Mean squared error 18.94 {'alpha': 0.031724137931034485}
Fold 11 Mean squared error 19.21 {'alpha': 0.035137931034482761}
Fold 12 Mean squared error 19.45 {'alpha': 0.038551724137931037}
Fold 13 Mean squared error 19.67 {'alpha': 0.041965517241379313}
Fold 14 Mean squared error 19.87 {'alpha': 0.045379310344827589}
Fold 15 Mean squared error 20.06 {'alpha': 0.048793103448275865}
Fold 16 Mean squared error 20.24 {'alpha': 0.052206896551724141}
Fold 17 Mean squared error 20.40 {'alpha': 0.055620689655172417}
Fold 18 Mean squared error 20.55 {'alpha': 0.059034482758620693}
Fold 19 Mean squared error 20.69 {'alpha': 0.062448275862068969}
Fold 20 Mean squared error 20.82 {'alpha': 0.065862068965517245}
Fold 21 Mean squared error 20.95 {'alpha': 0.069275862068965521}
Fold 22 Mean squared error 21.07 {'alpha': 0.072689655172413797}
Fold 23 Mean squared error 21.18 {'alpha': 0.076103448275862073}
Fold 24 Mean squared error 21.28 {'alpha': 0.079517241379310349}
Fold 25 Mean squared error 21.38 {'alpha': 0.082931034482758625}
Fold 26 Mean squared error 21.48 {'alpha': 0.086344827586206901}
Fold 27 Mean squared error 21.57 {'alpha': 0.089758620689655177}
Fold 28 Mean squared error 21.66 {'alpha': 0.093172413793103454}
Fold 29 Mean squared error 21.74 {'alpha': 0.09658620689655173}
Fold 30 Mean squared error 21.82 {'alpha': 0.10000000000000001}
```

This way, iteratively we arrived at the range starting at 0.0015 and ending at 0.0017.

We will then get the best estimator from our grid search and apply it to our train and test data:

```
choosen_model = build_model(x_train_poly,y_train)
predicted_y = choosen_model.predict(x_train_poly)
model_worth(y_train,predicted_y)
```

Our `model_worth` function prints the mean squared error value in our training dataset:

```
Mean squared error = 7.57
```

Let's view our coefficient weights:

```
Model coefficients
=====
Coefficient 1  0.000
Coefficient 2  0.145
Coefficient 3  -0.126
Coefficient 4  0.260
Coefficient 5  13.106
Coefficient 6  24.555
Coefficient 7  17.453
Coefficient 8  0.243
Coefficient 9  -2.255
Coefficient 10 0.822
Coefficient 11 0.022
Coefficient 12 0.812
Coefficient 13 0.052
Coefficient 14 0.696
Coefficient 15 0.300
Coefficient 16 0.010
Coefficient 17 2.792
Coefficient 18 -0.473
Coefficient 19 0.071
Coefficient 20 -0.001
Coefficient 21 -0.404
Coefficient 22 -0.005
Coefficient 23 0.000
Coefficient 24 0.007
Coefficient 25 -0.000
```

We have not displayed all of them but when you run the code, you can view all the values.

Finally, let's apply the model to our test dataset:

```
Mean squared error = 11.72
```

Thus, we used cross-validation and grid search to arrive at an alpha value for our ridge regression successfully. Our model has resulted in a lower mean squared error compared with the value in the ridge regression recipe.

There's more...

There are other cross-validation iterators available with scikit-learn. Of particular interest in this case is the leave-one-out iterator. You can read more about this at http://scikit-learn.org/stable/modules/cross_validation.html#leave-one-out-loo.

In this method, given the number of folds, it leaves one record to test and returns the rest to train. For example, if your input data has 100 instances and if we require five folds, we will get 99 instances to train and one to test in each fold.

In the grid search method that we used before, if we don't provide a custom iterator to the **cross-validation (cv)** parameter, it will by default use the leave-one-out cross-validation method:

```
grid = GridSearchCV(estimator=model, param_grid=grid_param,  
                    cv=None, scoring='mean_squared_error')
```

See also

- ▶ *Scaling the data recipe in Chapter 3, Data Analysis – Explore and Wrangle*
- ▶ *Standardizing the data recipe in Chapter 3, Data Analysis – Explore and Wrangle*
- ▶ *Preparing data for model building recipe in Chapter 6, Machine Learning I*
- ▶ *Regression with L2 Shrinkage – Ridge recipe in Chapter 7, Machine Learning II*
- ▶ *Regression with L2 Shrinkage – Lasso recipe in Chapter 7, Machine Learning II*

8

Ensemble Methods

In this chapter, we will look at the following recipes:

- ▶ Understanding Ensemble – the bagging Method
- ▶ Understanding Ensemble – the boosting Method, AdaBoost
- ▶ Understanding Ensemble – the gradient Boosting

Introduction

In this chapter, we are going to look at recipes covering the ensemble methods. When we are faced with the challenge of uncertainty while making a decision in real life, we typically approach multiple friends for opinion. We make our decision based on the collective knowledge that we receive from those friends. Ensemble is a similar concept in machine learning. In the previous chapters, we built a single model for our datasets, and used the prediction of that model on unseen test data. What if we build a lot of models on that data and make our final prediction based on the prediction from all these individual models? This is the idea behind Ensemble. Using the Ensemble approach for a given problem, we proceed with building a lot of models, and use all of them to make our final prediction on unseen datasets. For a regression problem, the final output may be the average prediction value from all the models. In a classification context, a majority vote is taken to decide the output class.

The fundamental idea is to have a lot of models, each one of them producing slightly different results on the training dataset. Some models learn certain aspects of the data very well as compared to the others. The belief is that the final output from all these models should be better than the output produced by just any one of them.

As mentioned earlier, the idea of ensemble is to combine many models together. These models can be of the same or of different types. For example, we can combine a neural network model output with a Bayesian model. We will restrict our discussions to using an ensemble of the same type of models in this chapter. Combining the same kind of models is vastly used in the Data Science community through techniques like Bagging and Boosting.

Bootstrap aggregation, commonly known as Bagging, is an elegant technique for generating a lot of models and combining their output to make a final prediction. Every model in a Bagging ensemble uses only a portion of the training data. The idea behind Bagging is to reduce the overfitting of data. As stated before, we want each of the model to be slightly different from the others. So we sample the data with replacement for training each of the models, and thus introduce variability. Another way to introduce variation in the model is to sample the attributes. We don't provide all the attributes to the model, but different models get a different set of attributes. Bagging can be easily parallelized. Based on the parallel processing framework available, models can be constructed in parallel with different samples of the training dataset. Bagging doesn't work with linear predictors like linear regression.

Boosting is an ensemble technique which produces a sequence of increasingly complex models. It works sequentially by training the newer models based on the errors in the previous models. Every model that is trained is associated with a weight, which is calculated based on how well the model has performed on the given data. When the final prediction is made, these weights decides the amount of influence that a particular model has over the final output. Boosting does not lend itself to parallelism as naturally as Bagging. Since the models are built in a sequence, it cannot be parallelized. Errors made by classifiers coming early in the sequence are considered as hard instances to classify. The framework is designed in such a way that models coming later in the sequence pick up those misclassified or erroneous predictions made by the previous predictor, and try to improve upon them. Typically, very weak classifiers are used in Boosting, for example a decision stump, which is a decision tree with a single splitting node and two leaves, is used inside the ensemble. A very famous success story about Boosting is the Viola-Jon Face Detection algorithm where several weak classifiers (decision stumps) were used to find good features. You can read more about this success story at the following website:

https://en.wikipedia.org/wiki/Viola%20%93Jones_object_detection_framework

In this chapter, we will study the Bagging and Boosting Methods in detail. We will extend our discussion to a special type of Boosting called Gradient Boosting in the final recipe. We will also take a look at both regression and classification problems, and see how they can be addressed by ensemble learning.

Understanding Ensemble – Bagging Method

Ensemble methods belong to the family of methods known as committee-based learning. Instead of leaving the decision of classification or regression to a single model, a group of models is used to make decisions in an ensemble. Bagging is a famous and widely used ensemble method.

Bagging is also known as bootstrap aggregation. Bagging can be made effective only if we are able to introduce variability in the underlying models, that is, if we can successfully introduce variability in the underlying dataset, it will lead to models with slight variations.

We leverage Bootstrapping to feed to these models variability in our dataset. Bootstrapping is the process by which we randomly sample the given dataset for a specified number of instances, with or without replacement. In bagging, we leverage bootstrapping to generate, say, m is the different datasets and construct a model for each of them. Finally, we average the output of all the models to produce the final prediction in case of regression problems.

Let us say we bootstrap the data m times, we would have m models, that is, y_m values, and our final prediction would be as follows:

$$Y_{final(x)} = \frac{1}{m} \sum_{i=1}^m y_m(x)$$

In case of classification problems, the final output is decided based on voting. Let us say we have one hundred models in our ensemble, and we have a two-class classification problem with class labels as $\{+1, -1\}$. If more than 50 models predict the output as $+1$, we declare the prediction as $+1$.

Randomization is another technique by which variability can be introduced in the model building exercise. An example is to pick randomly a subset of attributes for each model in the ensemble. That way, different models will have different sets of attributes. This technique is called the random subspaces method.

With very stable models, Bagging may not achieve very great results. Bagging helps most if the underlying classifier is very sensitive to even small changes to the data. For example, Decision trees, which are very unstable. Unpruned decision trees are a good candidate for Bagging. But say a Nearest Neighbor Classifier, K, is a very stable model. However, we can leverage the random subspaces, and introduce some instability into the nearest neighbor methods.

In the following recipe, you will learn how to leverage Bagging and Random subspaces on a K-Nearest Neighbor algorithm. We will take up a classification problem, and the final prediction will be based on majority voting.

Getting ready...

We will leverage the Scikit learn classes' `KNeighborsClassifier` for classification and `BaggingClassifier` for applying the bagging principle. We will generate data for this recipe using the `make_classification` convenience function.

How to do it

Let us import the necessary libraries, and write a function `get_data()` to provide us with a dataset to work through this recipe:

```
from sklearn.datasets import make_classification
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.metrics import classification_report
from sklearn.cross_validation import train_test_split

def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 30
    redundant_features = int(0.1*no_features)
    informative_features = int(0.6*no_features)
    repeated_features = int(0.1*no_features)
    print no_features, redundant_features, informative_
    features, repeated_features
    x,y = make_classification(n_samples=500,n_features=no_
    features,flip_y=0.03,\n        n_informative = informative_features, n_redundant =
    redundant_features \
        ,n_repeated = repeated_features,random_state=7)
    return x,y
```

Let us proceed to write three functions:

Function `build_single_model` to make a simple KNearest neighbor model with the given data.

Function `build_bagging_model`, a function which implements the Bagging routine.

The function `view_model` to inspect the model that we have built:

```
def build_single_model(x,y):
    model = KNeighborsClassifier()
    model.fit(x,y)
    return model

def build_bagging_model(x,y):
    bagging = BaggingClassifier(KNeighborsClassifier(),n_
estimators=100,random_state=9 \
,max_samples=1.0,max_features=0.7,bootstrap=True,bootstr\
ap_features=True)
    bagging.fit(x,y)
    return bagging

def view_model(model):
    print "\n Sampled attributes in top 10 estimators\n"
    for i,feature_set in enumerate(model.estimators_features_[0:10]):
        print "estimator %d"%(i+1),feature_set
```

Finally, we will write our main function, which will call the other functions:

```
if __name__ == "__main__":
    x,y = get_data()

    # Divide the data into Train, dev and test
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)

    # Build a single model
    model = build_single_model(x_train,y_train)
    predicted_y = model.predict(x_train)
    print "\n Single Model Accuracy on training data\n"
    print classification_report(y_train,predicted_y)
    # Build a bag of models
    bagging = build_bagging_model(x_train,y_train)
    predicted_y = bagging.predict(x_train)
    print "\n Bagging Model Accuracy on training data\n"
    print classification_report(y_train,predicted_y)
    view_model(bagging)

    # Look at the dev set
```

```
predicted_y = model.predict(x_dev)
print "\n Single Model Accuracy on Dev data\n"
print classification_report(y_dev,predicted_y)

print "\n Bagging Model Accuracy on Dev data\n"
predicted_y = bagging.predict(x_dev)
print classification_report(y_dev,predicted_y)
```

How it works...

Let us start with the main method. We first call the `get_data` function to return the dataset as a matrix `x` of predictors and a vector `y` for the response variable. Let us look into the `get_data` function:

```
no_features = 30
redundant_features = int(0.1*no_features)
informative_features = int(0.6*no_features)
repeated_features = int(0.1*no_features)
x,y =make_classification(n_samples=500,n_features=no_features,flip_
y=0.03,\n_informative = informative_features, n_redundant = redundant_features
\n_repeated = repeated_features,random_state=7)
```

Take a look at the parameters passed to the `make_classification` method. The first parameter is the number of instances required; in this case, we say we need 500 instances. The second parameter is the number of attributes that are required per instance. We say that we need 30 of them as defined by the variable `no_features`. The third parameter, `flip_y`, randomly interchanges 3 percent of the instances. This is done to introduce some noise in our data. The next parameter specifies the number of features out of those 30 that should be informative enough to be used in our classification. We have specified that 60 percent of our features, that is, 18 out of 30 should be informative. The next parameter is about redundant features. These are generated as a linear combination of the informative features in order to introduce a correlation among the features. Finally, repeated features are the duplicate features which are drawn randomly from both informative features and redundant features.

Let us split the data into a training and a testing set using `train_test_split`. We reserve 30 percent of our data for testing:

```
# Divide the data into Train, dev and test
x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
```

Once again we leverage `train_test_split` to split our test data into dev and test.

```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_all,test_size=0.3,random_state=9)
```

Having divided the data for building, evaluating, and testing the model, we proceed to build our models. We are going to initially build a single model using `KNeighborsClassifier` by invoking the following:

```
model = build_single_model(x_train,y_train)
```

Inside this function, we create an object of type `KNeighborsClassifier` and fit our data, as follows:

```
def build_single_model(x,y):
    model = KNeighborsClassifier()
    model.fit(x,y)
    return model
```

As explained in the previous section, `KNearestNeighbor` is a very stable algorithm. Let us see how this model performs. We perform our predictions on the training data and look at our model metrics:

```
predicted_y = model.predict(x_train)
print "\n Single Model Accuracy on training data\n"
print classification_report(y_train,predicted_y)
```

`classification_report` is a convenient function under the module `metric` in Scikit learn. It gives a table for precision, recall, and f1-score:

Single Model Accuracy on training data				
	precision	recall	f1-score	support
0	0.88	0.87	0.88	181
1	0.87	0.88	0.87	169
avg / total	0.87	0.87	0.87	350

Out of 350 instances, our precision is 87 percent. With this figure, let us proceed to build our bagging model:

```
bagging = build_bagging_model(x_train,y_train)
```

We invoke the function `build_bagging_model` with our training data to build a bag of classifiers, as follows:

```
def build_bagging_model(x,y):
    bagging = BaggingClassifier(KNeighborsClassifier(),n_
estimators=100,random_state=9 \
,max_samples=1.0,max_features=0.7,bootstrap=True,bootstr\
ap_features=True)
    bagging.fit(x,y)
    return bagging
```

Inside the method, we invoke the `BaggingClassifier` class. Let us look at the arguments that we pass to this class to initialize it.

The first argument is the underlying estimator or model. By passing `KNeighorClassifier`, we are telling the bagging classifier that we want to build a bag of `KNearestNeighbor` classifiers. The next parameter specifies the number of estimators that we will build. In this case, we are saying we need 100 of them. The `random_state` argument is the seed to be used by the random number generator. In order to be consistent during different runs, we set this to an integer value.

Our next parameter is `max_samples`, where we specify the number of instances to be selected for one estimator when we bootstrap from our input dataset. In this case, we are asking the bagging routine to select all the instances.

Next, the parameter `max_features` specifies the number of attributes that are to be included while bootstrapping for an estimator. We say that we want to include only 70 percent of the attributes. So for each estimator/model inside the ensemble, it will be using a different subset of the attributes to build the model. This is the random space methodology that we introduced in the previous section. The function proceeds to fit the model and return the model to the calling function.

```
bagging = build_bagging_model(x_train,y_train)
predicted_y = bagging.predict(x_train)
print "\n Bagging Model Accuracy on training data\n"
print classification_report(y_train,predicted_y)
```

Let us look at the model accuracy:

Bagging Model Accuracy on training data				
	precision	recall	f1-score	support
0	0.94	0.97	0.95	181
1	0.96	0.93	0.95	169
avg / total	0.95	0.95	0.95	350

You can see a big jump in the model metrics.

Before we test our models with our dev dataset, let us look at the attributes that were allocated to the different models, by invoking the view_model function:

```
view_model(bagging)
```

We print the attributes selected for the first ten models, as follows:

```
def view_model(model):  
    print "\n Sampled attributes in top 10 estimators\n"  
    for i,feature_set in enumerate(model.estimators_features_[0:10]):  
        print "estimator %d"% (i+1), feature_set
```

```
Sampled attributes in top 10 estimators  
  
estimator 1 [20 10 6 17 18 11 17 9 14 3 10 10 23 22 18 17 11 21 20 1 16]  
estimator 2 [3 27 28 20 20 27 25 0 21 1 12 20 21 29 1 0 28 16 4 9 10]  
estimator 3 [5 23 19 2 16 21 4 13 27 1 15 24 5 14 1 4 25 22 26 29 15]  
estimator 4 [23 10 16 7 22 11 0 14 14 17 8 17 27 12 13 23 8 7 27 0 27]  
estimator 5 [0 26 13 23 7 27 15 18 11 26 18 26 3 22 6 11 21 6 12 19 7]  
estimator 6 [5 24 19 21 2 2 22 12 21 14 28 5 29 9 19 24 14 21 8 11 26]  
estimator 7 [23 2 17 22 2 12 14 25 5 7 10 25 5 17 16 9 0 9 9 15 4]  
estimator 8 [10 7 8 8 18 6 3 12 29 13 17 20 9 2 25 6 28 15 0 16 20]  
estimator 9 [29 2 5 6 11 18 4 19 27 17 28 20 15 21 26 14 5 28 15 21 26]  
estimator 10 [22 17 10 16 10 27 8 2 18 26 1 3 2 1 17 2 12 10 22 26 27]
```

As you can make out from the result, we have assigned attributes to every estimator pretty much randomly. In this way, we introduced variability into each of our estimator.

Let us proceed to check how our single classifier and bag of estimators have performed in our dev set:

```
# Look at the dev set
predicted_y = model.predict(x_dev)
print "\n Single Model Accuracy on Dev data\n"
print classification_report(y_dev,predicted_y)

print "\n Bagging Model Accuracy on Dev data\n"
predicted_y = bagging.predict(x_dev)
print classification_report(y_dev,predicted_y)
```

Single Model Accuracy on Dev data				
	precision	recall	f1-score	support
0	0.83	0.84	0.83	51
1	0.85	0.83	0.84	54
avg / total	0.84	0.84	0.84	105
Bagging Model Accuracy on Dev data				
	precision	recall	f1-score	support
0	0.85	0.88	0.87	51
1	0.88	0.85	0.87	54
avg / total	0.87	0.87	0.87	105

As expected, our bag of estimators has performed better in our dev set as compared to our single classifier.

There's more...

As we said earlier, in the case of classification, the label with the majority number of votes is considered as the final prediction. Instead of the voting scheme, we can ask the constituent models to output the prediction probabilities for the labels. An average of the probabilities can be finally taken to decide the final output label. In Scikit's case, the documentation of the API provides the details on how the final prediction is performed:

'The predicted class of an input sample is computed as the class with the highest mean predicted probability. If base estimators do not implement a predict_proba method, then it resorts to voting.'

<http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html>

In the last chapter, we discussed cross validation. Although cross validation may look very similar to bagging, they have different uses in reality. In cross validation, we create K-Folds and, based on the model output from those folds, we may choose our parameters for the model, as we selected the alpha value for ridge regression. This is done primarily to avoid exposing our test data in the model building exercise. Cross validation can be used in Bagging to determine the number of estimators we need to add to our bagging module.

However, a drawback with Bagging is that we loose the interpretability of a model. Consider a Simple Decision tree derived after pruning. It is very easy to explain the decision tree model. But once we have a bag of 100 such models, it become a black box. For increased accuracy, we do a trading of interpretability.

Please refer to the following paper by Leo Breiman for more information about bagging:

Leo Breiman. 1996. *Bagging predictors*. *Mach. Learn.* 24, 2 (August 1996), 123-140. DOI=10.1023/A:1018054314350 <http://dx.doi.org/10.1023/A:1018054314350>

See also

- ▶ *Using cross validation iterators* recipe in Chapter 7, Machine Learning 2
- ▶ *Building Decision Trees to solve Multi-Class Problems* recipe in Chapter 6, Machine Learning 1

Understanding Ensemble – Boosting Method

Boosting is a powerful ensemble technique. It's pretty much used in most of the Data Science applications. In fact, it is one of the most essential tools in a Data Scientist tool kit. The Boosting technique utilizes a bag of estimators similar to Bagging. But that is where the similarity ends. Let us quickly see how Boosting acts as a very effective ensemble technique before jumping into our recipe.

Let's take the familiar two-class classification problem, where the input is a set of predictors (X) and the output is a response variable (Y) which can either take 0 or 1 as value. The input for the classification problem is represented as follows:

$$X = \{x_1, x_2, \dots, x_N\} \text{ and } Y = \{0, 1\}$$

The job the classifier is to find a function which can approximate:

$$Y = F(X)$$

The misclassification rate of the classifier is defined as:

$$\text{error rate} = \frac{1}{N} \sum_{i=1}^N \text{instance where } y_i \neq F(x_i)$$

Let us say we build a very weak classifier whose error rate is slightly better than random guessing. In Boosting, we build a sequence of weak classifiers on a slightly modified set of data. We modify the data slightly for every classifier, and finally, we end up with M classifiers:

$$F_1(X), F_2(X), \dots, F_M(X)$$

Finally, the predictions from all of them are combined through a weighted majority vote:

$$F_{final}(X) = sign\left(\sum_{i=1}^M \alpha_i F_i(X)\right)$$

This method is called AdaBoost.

The weight alpha and the sequential way of model building is where Boosting differs from Bagging. As mentioned earlier, Boosting builds a sequence of weak classifiers on a slightly modified data set for each classifier. Let us look at what that slight data modification refers to. It is from this modification that we derive our weight alpha.

Initially for the first classifier, m=1, we set the weight of each instance as 1/N, that is, if there are a hundred records, each record gets a weight of 0.001. Let us denote the weight by w—now we have a hundred such weights:

$$w_1, w_2, \dots, w_N$$

All the records now have an equal chance of being selected by a classifier. We build the classifier, and test it against our training data to get the misclassification rate. Refer to the misclassification rate formula given earlier in this section. We are going to change it slightly by including the weights, as follows:

$$\text{error rate}_1 = \frac{\sum_{i=1}^N w_i * \text{abs}(y_i - \text{predicted } y_i)}{\sum_{i=1}^N w_i}$$

Where abs stands for the absolute value of the results. With this error rate, we calculate our alpha (model weight) as follows:

$$\alpha_1 = 0.5 * \log\left(\frac{(1 - \text{error rate}_1 + \text{epsilon})}{\text{error rate}_1 + \text{epsilon}}\right)$$

Where epsilon is a very small value.

Let us say our model 1 has got an error rate of 0.3, that is, the model was able to classify 70 percent of the records correctly. Therefore, the weight for that model will be 0.8, approximately, which, is a good weight. Based on this, we go will back and set the weights of individual records, as follows:

$$w_i = w_i * \exp(\alpha_1 * \text{abs}(y_i - \text{predicted } y_i))$$

As you can see, the weights of all the attributes which were misclassified will increase. This increases the chance of the misclassified record being selected by the next classifier. Thus, the classifier coming next in the sequence selects the instances with more weight and tries to fit it. In this way, all the future classifiers start concentrating on the records misclassified by the previous classifier.

This is the power of boosting. It is able to turn several weak classifiers into one powerful ensemble.

Let us see boosting in action. As we proceed with our code, we will also see a small variation to AdaBoost known as SAMME.

Getting Started...

We will leverage the scikit learn classes `DecisionTreeClassifier` for classification and the `AdaBoostClassifier` for applying the Boosting principle. We will generate data for this recipe using the `make_classification` convenience function.

How to do it

Let us import the necessary libraries, and write a function `get_data()` to provide us with a dataset to work through this recipe.

```
from sklearn.datasets import make_classification
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import classification_report, zero_one_loss
from sklearn.cross_validation import train_test_split
from sklearn.tree import DecisionTreeClassifier
import numpy as np
import matplotlib.pyplot as plt
import itertools

def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 30
    redundant_features = int(0.1*no_features)
    informative_features = int(0.6*no_features)
    repeated_features = int(0.1*no_features)
    print no_features, redundant_features, informative_
    features, repeated_features
    x,y = make_classification(n_samples=500,n_features=no_
    features, flip_y=0.03,\n        n_informative = informative_features, n_redundant =
    redundant_features \
        ,n_repeated = repeated_features,random_state=7)
    return x,y

def build_single_model(x,y):
    model = DecisionTreeClassifier()
    model.fit(x,y)
    return model

def build_boosting_model(x,y,no_estimators=20):
    boosting = AdaBoostClassifier(DecisionTreeClassifier(max_
    depth=1,min_samples_leaf=1),random_state=9 \
        ,n_estimators=no_estimators,algorithm="SAMME")
    boosting.fit(x,y)
    return boosting

def view_model(model):
```

```
print "\n Estimator Weights and Error\n"
for i,weight in enumerate(model.estimator_weights_):
    print "estimator %d weight = %0.4f error =
%0.4f"%(i+1,weight,model.estimator_errors_[i])

plt.figure(1)
plt.title("Model weight vs error")
plt.xlabel("Weight")
plt.ylabel("Error")
plt.plot(model.estimator_weights_,model.estimator_errors_)

def number_estimators_vs_err_rate(x,y,x_dev,y_dev):
    no_estimators = range(20,120,10)
    misclassy_rate = []
    misclassy_rate_dev = []

    for no_estimator in no_estimators:
        boosting = build_boosting_model(x,y,no_estimators=no_
estimator)
        predicted_y = boosting.predict(x)
        predicted_y_dev = boosting.predict(x_dev)
        misclassy_rate.append(zero_one_loss(y,predicted_y))
        misclassy_rate_dev.append(zero_one_loss(y_dev,predicted_y_
dev))

    plt.figure(2)
    plt.title("No estimators vs Mis-classification rate")
    plt.xlabel("No of estimators")
    plt.ylabel("Mis-classification rate")
    plt.plot(no_estimators,misclassy_rate,label='Train')
    plt.plot(no_estimators,misclassy_rate_dev,label='Dev')

    plt.show()

if __name__ == "__main__":
    x,y = get_data()
    plot_data(x,y)

    # Divide the data into Train, dev and test
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
```

```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_all,test_size=0.3,random_state=9)

# Build a single model
model = build_single_model(x_train,y_train)
predicted_y = model.predict(x_train)
print "\n Single Model Accuracy on training data\n"
print classification_report(y_train,predicted_y)
print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_train,predicted_y)*100), "%"

# Build a bag of models
boosting = build_boosting_model(x_train,y_train, no_estimators=85)
predicted_y = boosting.predict(x_train)
print "\n Boosting Model Accuracy on training data\n"
print classification_report(y_train,predicted_y)
print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_train,predicted_y)*100), "%"

view_model(boosting)

# Look at the dev set
predicted_y = model.predict(x_dev)
print "\n Single Model Accuracy on Dev data\n"
print classification_report(y_dev,predicted_y)
print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_dev,predicted_y)*100), "%"

print "\n Boosting Model Accuracy on Dev data\n"
predicted_y = boosting.predict(x_dev)
print classification_report(y_dev,predicted_y)
print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_dev,predicted_y)*100), "%"

number_estimators_vs_err_rate(x_train,y_train,x_dev,y_dev)
```

Let us proceed and write the following three functions:

The function `build_single_model` to make a simple Decision Tree model with the given data.

The function `build_boosting_model`, a function which implements the Boosting routine.

The function `view_model`, to inspect the model that we have built.

```
def build_single_model(x,y):
    model = DecisionTreeClassifier()
    model.fit(x,y)
    return model

def build_boosting_model(x,y,no_estimators=20):
    boosting = AdaBoostClassifier(DecisionTreeClassifier(max_
depth=1,min_samples_leaf=1),random_state=9 \
    ,n_estimators=no_estimators,algorithm="SAMME")
    boosting.fit(x,y)
    return boosting

def view_model(model):
    print "\n Estimator Weights and Error\n"
    for i,weight in enumerate(model.estimator_weights_):
        print "estimator %d weight = %0.4f error =
%0.4f"%(i+1,weight,model.estimator_errors_[i])

    plt.figure(1)
    plt.title("Model weight vs error")
    plt.xlabel("Weight")
    plt.ylabel("Error")
    plt.plot(model.estimator_weights_,model.estimator_errors_)
```

We then write a function called `number_estimators_vs_err_rate`. We use this function to see how our error rates change with respect to the number of models in our ensemble.

```
def number_estimators_vs_err_rate(x,y,x_dev,y_dev):
    no_estimators = range(20,120,10)
    misclassy_rate = []
    misclassy_rate_dev = []

    for no_estimator in no_estimators:
        boosting = build_boosting_model(x,y,no_estimators=no_
estimator)
        predicted_y = boosting.predict(x)
        predicted_y_dev = boosting.predict(x_dev)
        misclassy_rate.append(zero_one_loss(y,predicted_y))
        misclassy_rate_dev.append(zero_one_loss(y_dev,predicted_y_
dev))

    plt.figure(2)
    plt.title("No estimators vs Mis-classification rate")
    plt.xlabel("No of estimators")
```

```
plt.ylabel("Mis-classification rate")
plt.plot(no_estimators,misclassy_rate,label='Train')
plt.plot(no_estimators,misclassy_rate_dev,label='Dev')

plt.show()
```

Finally, we will write our main function, which will call the other functions.

```
if __name__ == "__main__":
    x,y = get_data()
    plot_data(x,y)

    # Divide the data into Train, dev and test
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_-
all,test_size=0.3,random_state=9)

    # Build a single model
    model = build_single_model(x_train,y_train)
    predicted_y = model.predict(x_train)
    print "\n Single Model Accuracy on training data\n"
    print classification_report(y_train,predicted_y)
    print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_
train,predicted_y)*100), "%"

    # Build a bag of models
    boosting = build_boosting_model(x_train,y_train, no_estimators=85)
    predicted_y = boosting.predict(x_train)
    print "\n Boosting Model Accuracy on training data\n"
    print classification_report(y_train,predicted_y)
    print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_
train,predicted_y)*100), "%"

    view_model(boosting)

    # Look at the dev set
    predicted_y = model.predict(x_dev)
    print "\n Single Model Accuracy on Dev data\n"
    print classification_report(y_dev,predicted_y)
    print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_
dev,predicted_y)*100), "%"

    print "\n Boosting Model Accuracy on Dev data\n"
```

```
predicted_y = boosting.predict(x_dev)
print classification_report(y_dev,predicted_y)
print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_
dev,predicted_y)*100), "%"

number_estimators_vs_err_rate(x_train,y_train,x_dev,y_dev)
```

How it works...

Let us start with the main method. We first call the `get_data` function to return the dataset as a matrix `x` of predictors and a vector `y` for the response variable. Let us look into the `get_data` function:

```
no_features = 30
redundant_features = int(0.1*no_features)
informative_features = int(0.6*no_features)
repeated_features = int(0.1*no_features)

x,y = make_classification(n_samples=500,n_features=no_features,flip_
y=0.03,\n_informative = informative_features, n_redundant = redundant_features
\n_repeated = repeated_features,random_state=7)
```

Take a look at the parameters passed to the `make_classification` method. The first parameter is the number of instances required; in this case, we say we need 500 instances. The second parameter gives the number of attributes that are required per instance. We say that we need 30 of them, as defined by the variable `no_features`. The third parameter `flip_y`, randomly interchanges 3 percent of the instances. This is done to introduce some noise in our data. The next parameter specifies the number of features out of those 30 which should be informative enough to be used in our classification. We have specified that 60 percent of our features, that is, 18 out of 30 should be informative. The next parameter is about redundant features. These are generated as a linear combination of the informative features for introducing a correlation among the features. Finally, repeated features are the duplicate features which are drawn randomly from both, informative features and redundant features.

Let us split the data into a training and a testing set using `train_test_split`. We reserve 30 percent of our data for testing.

```
# Divide the data into Train, dev and test
x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
```

Once again, we leverage `train_test_split` to split our test data into dev and test.

```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)
```

Having divided the data for building, evaluating, and testing the model, we proceed to build our models.

Let us start by fitting a single decision tree, and look at the performance of the tree on training set:

```
# Build a single model
model = build_single_model(x_train,y_train)
```

We build a model by invoking `build_single_model` function with the predictors and response variable. Inside this we fit a single decision tree and return the tree back to the calling function.

```
def build_single_model(x,y):
    model = DecisionTreeClassifier()
    model.fit(x,y)
    return model
```

Let us evaluate how good the model is using `classification_report`, a utility function from Scikit learn which displays a set of metrics including precision, recall and f1-score; we also display the misclassification rate.

```
predicted_y = model.predict(x_train)
print "\n Single Model Accuracy on training data\n"
print classification_report(y_train,predicted_y)
print "Fraction of misclassification =
%0.2f"%(zero_one_loss(y_train,predicted_y)*100), "%"
```

Single Model Accuracy on training data				
	precision	recall	f1-score	support
0	1.00	1.00	1.00	181
1	1.00	1.00	1.00	169
avg / total	1.00	1.00	1.00	350

Fraction of misclassification = 0.00 %

As you can see, our decision tree model has done a perfect job of fitting the data—our misclassification rate is 0. Before we test this model on our dev data, let us build our ensemble:

```
# Build a bag of models
boosting = build_boosting_model(x_train,y_train, no_estimators=85)
```

Using the method `build_boosting_model`, we build our ensemble as follows:

```
boosting = AdaBoostClassifier(DecisionTreeClassifier(max_
depth=1,min_samples_leaf=1),random_state=9 \
,n_estimators=no_estimators,algorithm="SAMME")
boosting.fit(x,y)
```

We leverage `AdaBoostClassifier` from Scikit learn to build our Boosting ensemble. We instantiate the class with the following parameters:

An estimator—in our case, we say we want to build an ensemble of decision trees. Hence, we pass the `DecisionTreeClassifier` object.

`max_depth`—We don't want to have fully grown trees in our ensemble. We need only stumps—trees with just two leaf nodes and one splitting node. Hence, we set the `max_depth` parameter to 1.

With the `n_estimators` parameters, we specify the number of trees that we want to grow; in this case, we will grow 86 trees.

Finally, we have a parameter called `algorithm`, which is set to `SAMME`. `SAMME` stands for Stage wise Additive Modeling using Multi-class Exponential loss function. `SAMME` is an improvement over the `AdaBoosting` algorithm. It tries to put more weights on the misclassified records. The model weight alpha is where `SAMME` differs from `AdaBoost`.

$$\alpha_i = \log\left(\frac{(1 - \text{error rate}_i + \epsilon)}{\text{error rate}_i + \epsilon}\right) + \log(K-1)$$

We have ignored the constant 0.5 in the preceding formula. Let us look at the new addition: $\log(K-1)$. If $K = 2$, then the preceding equation reduces to `AdaBoost`. Here, K is the number of classes in our response variable. For a two-class problem, `SAMME` reduces to `AdaBoost`, as stated earlier.

Let us fit the model, and return it to our calling function. We run this model on our training dataset, and once again look at how the model has performed:

```
predicted_y = boosting.predict(x_train)
print "\n Boosting Model Accuracy on training data\n"
print classification_report(y_train,predicted_y)
```

```
print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_train,predicted_y)*100), "%"
```

Boosting Model Accuracy on training data				
	precision	recall	f1-score	support
0	0.98	0.98	0.98	181
1	0.98	0.98	0.98	169
avg / total	0.98	0.98	0.98	350

Fraction of misclassification = 1.71 %

The result is not very different from our original model. We have correctly classified almost 98 percent of the records with this ensemble.

Before testing it on our dev set, let us proceed to look at the Boosting ensemble that we have built:

```
view_model(boosting)
```

Inside view_model, we first print out the weights assigned to each classifier in our ensemble:

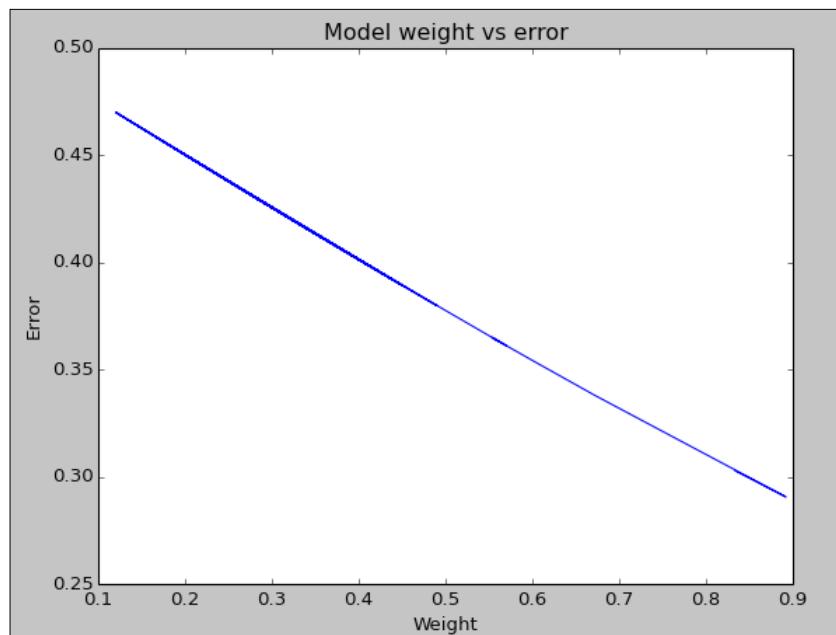
```
print "\n Estimator Weights and Error\n"
for i,weight in enumerate(model.estimator_weights_):
    print "estimator %d weight = %0.4f error =
%0.4f"%(i+1,weight,model.estimator_errors_[i])
```

Estimator Weights and Error	
estimator 1 weight = 0.8337	error = 0.3029
estimator 2 weight = 0.8921	error = 0.2907
estimator 3 weight = 0.6730	error = 0.3378
estimator 4 weight = 0.6067	error = 0.3528
estimator 5 weight = 0.5746	error = 0.3602
estimator 6 weight = 0.5537	error = 0.3650
estimator 7 weight = 0.5697	error = 0.3613
estimator 8 weight = 0.5538	error = 0.3650
estimator 9 weight = 0.5579	error = 0.3640
estimator 10 weight = 0.4530	error = 0.3886
estimator 11 weight = 0.4530	error = 0.3886
estimator 12 weight = 0.3564	error = 0.4118
estimator 13 weight = 0.4130	error = 0.3982
estimator 14 weight = 0.3679	error = 0.4091
estimator 15 weight = 0.3142	error = 0.4221
estimator 16 weight = 0.3888	error = 0.4040
estimator 17 weight = 0.4902	error = 0.3799
estimator 18 weight = 0.2798	error = 0.4305
estimator 19 weight = 0.4463	error = 0.3902
estimator 20 weight = 0.2645	error = 0.4343

Here we have shown the weights of the first 20 ensembles. Based on their misclassification rate, we have assigned different weights to these estimators.

Let us proceed to plot a graph showing the estimator weight versus the error thrown by each estimator:

```
plt.figure(1)
plt.title("Model weight vs error")
plt.xlabel("Weight")
plt.ylabel("Error")
plt.plot(model.estimator_weights_,model.estimator_errors_)
```



As you can see, the models which are classifying properly are assigned more weights than the ones with higher errors.

Let us now look at the way single tree and the bag of tree have performed against the dev data:

```
# Look at the dev set
predicted_y = model.predict(x_dev)
print "\n Single Model Accuracy on Dev data\n"
print classification_report(y_dev,predicted_y)
```

```

print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_
dev,predicted_y)*100), "%"

print "\n Boosting Model Accuracy on Dev data\n"
predicted_y = boosting.predict(x_dev)
print classification_report(y_dev,predicted_y)
print "Fraction of misclassification = %0.2f"%(zero_one_loss(y_
dev,predicted_y)*100), "%"

```

Pretty much as we did with the training data, we print the classification report and the misclassification rate:

Single Model Accuracy on Dev data				
	precision	recall	f1-score	support
0	0.59	0.73	0.65	51
1	0.67	0.52	0.58	54
avg / total	0.63	0.62	0.62	105
Fraction of misclassification = 38.10 %				
Boosting Model Accuracy on Dev data				
	precision	recall	f1-score	support
0	0.77	0.86	0.81	51
1	0.85	0.76	0.80	54
avg / total	0.81	0.81	0.81	105
Fraction of misclassification = 19.05 %				

As you can see, the single tree has performed poorly. Though it displayed a 100 percent accuracy with the training data, with the dev data it has misclassified almost 40 percent of the records—a sign of overfitting. In contrast, the Boosting model is able to make a better fit of the dev data.

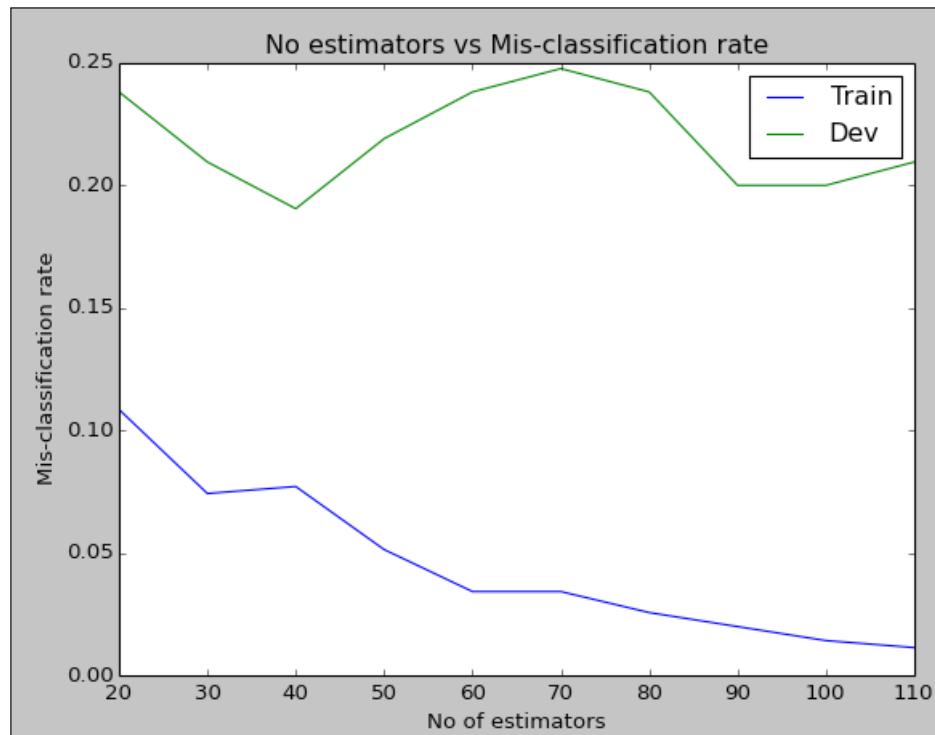
How do we go about improving the Boosting model? One way to do it is to test the error rate in the training set against the number of ensembles that we want to include in our bagging.

```
number_estimators_vs_err_rate(x_train,y_train,x_dev,y_dev)
```

The following function proceeds to fit with the increasing number of ensembles and plot the error rate:

```
def number_estimators_vs_err_rate(x,y,x_dev,y_dev):  
    no_estimators = range(20,120,10)  
    misclassy_rate = []  
    misclassy_rate_dev = []  
  
    for no_estimator in no_estimators:  
        boosting = build_boosting_model(x,y,no_estimators=no_  
estimator)  
        predicted_y = boosting.predict(x)  
        predicted_y_dev = boosting.predict(x_dev)  
        misclassy_rate.append(zero_one_loss(y,predicted_y))  
        misclassy_rate_dev.append(zero_one_loss(y_dev,predicted_y_  
dev))  
  
    plt.figure(2)  
    plt.title("No estimators vs Mis-classification rate")  
    plt.xlabel("No of estimators")  
    plt.ylabel("Mis-classification rate")  
    plt.plot(no_estimators,misclassy_rate,label='Train')  
    plt.plot(no_estimators,misclassy_rate_dev,label='Dev')  
  
    plt.show()
```

As you can see, we declare a list, starting with 20 and ending with 120 in step size of 10s. Inside the `for` loop, we pass each element of this list as the number of estimator parameter to `build_boosting_model`, and then proceed to access the error rate of the model. We then check the error rates in the dev set. Now we have two lists—one which has all the error rates from the training data and another with the error rates from the dev data. We plot them both, where the x axis is the number of estimators and y axis is the misclassification rate in the dev and train sets.



The preceding plot gives a clue that in around 30 to 40 estimators, the error rate in dev is very low. We can further experiment with the tree model parameters to arrive at a good model.

There's more...

Boosting was introduced in the following seminal paper:

Freund, Y. & Schapire, R. (1997), 'A decision theoretic generalization of on-line learning and an application to boosting', Journal of Computer and System Sciences 55(1), 119–139.

Initially, most of the Boosting methods reduced the multiclass problems into two-class problems and multiple two-class problems. The following paper extends AdaBoost to the multiclass problems:

Multi-class AdaBoost Statistics and Its Interface, Vol. 2, No. 3. (2009), pp. 349-360, doi:10.4310/sii.2009.v2.n3.a8 by Trevor Hastie, Saharon Rosset, Ji Zhu, Hui Zou

This paper also introduces SAMME, the method that we have used in our recipe.

See also

- ▶ *Building Decision Trees to solve Multi-Class Problems* recipe in Chapter 6, Machine Learning I
- ▶ *Using cross validation iterators* recipe in Chapter 7, Machine Learning II
- ▶ *Understanding Ensemble – Bagging Method* recipe in Chapter 8, Model Selection and Evaluation

Understanding Ensemble – Gradient Boosting

Let us recall the Boosting algorithm explained in the previous recipe. In boosting, we fitted an additive model in a forward, stage-wise manner. We built the classifiers sequentially. After building each classifier, we estimated the weight/importance of the classifiers. Based on weights/importance, we adjusted the weights of the instances in our training set. Misclassified instances were weighted higher than the correctly classified ones. We would like the next model to pick those incorrectly classified instances and train on them. Instances from the dataset which didn't fit properly were identified using these weights. Another way of looking at it is that those records were the shortcomings of the previous model. The next model tries to overcome those shortcomings.

Gradient Boosting uses gradients instead of weights to identify those shortcomings. Let us quickly see how we can use gradients to improve models.

Let us take a simple regression problem, where we are given the required predictor variable X and the response Y , which is a real number.

$$X = \{x_1, x_2, \dots, x_N\} \text{ and } Y = \{y_1, y_2, \dots, y_N\}$$

Gradient boosting proceeds as follows:

It starts with a very simple model, say, mean value.

$$\hat{y} = \frac{1}{N} \sum_{i=1}^N y_i$$

The predicted value is simply the mean value of the response variables.

It then proceeds to fit the residuals. Residual is the difference between the actual value y and the predicted value \hat{y} .

$$R_1 = y - \hat{y}$$

The next classifier is trained on the data set as follows:

$$\{(x_1, R_{11}), (x_2, R_{12}), \dots, (x_N, R_{1N})\}$$

The subsequent model is trained on the residual of the previous model, and thus, the algorithm proceeds to build the required number of models inside the ensemble.

Let us try and understand why we train on residuals. By now it should be clear that Boosting makes additive models. Let us say we build two models $F_1(X)$ and $F_2(X)$ to predict y_1 . By the additive principle, we can combine these two models as follows:

$$F_1(X) + F_2(X) = Y_1$$

That is, we combine the prediction from both the models to predict Y_1 .

Equivalently, we can say that:

$$F_2(X) = Y_1 - F_1(X)$$

$Y_1 - F_1(X)$ is the residual

Residual is the part where the model has not done well, or to put it simply, residuals are the shortcomings of the previous model. Hence, we use the residual to improve our model, that is, improving the shortcomings of the previous model. Based on this discussion, you would wonder why the method is called Gradient Boosting instead of Residual Boosting.

Given a function which is differentiable, Gradient stands for the first-order derivatives of that function at certain values. In the case of regression, the objective function is:

$$\frac{1}{N} \sum_{i=1}^N (y_i - F(x_i))^2$$

where $F(x_i)$ is our regression model.

The linear regression problem is about minimizing this preceding function. We find the first-order derivative of that function at value $F(x_i)$, and if we update our weights' coefficients with the negative of that derivative value, we will move towards a minimum solution in the search space. The first-order derivative of the preceding cost function with respect to $F(x_i)$ is $F'(x_i) - y_i$. Please refer to the following link for the derivation:

https://en.wikipedia.org/wiki/Gradient_descent

$F'(x_i) - y_i$, the gradient, is the negative of our residual $y_i - F(x_i)$, and hence the name Gradient Boosting.

With this theory in mind, let us jump into our recipe for gradient boosting.

Getting Started...

We are going to use the Boston dataset to demonstrate Gradient Boosting. The Boston data has 13 attributes and 506 instances. The target variable is a real number, the median value of houses in thousands. the Boston data can be downloaded from the UCI link:

<https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names>

We intend to generate Polynomial Features of degree 2, and consider only the interaction effects.

How to do it

Let us import the necessary libraries and write a function `get_data()` to provide us with a dataset to work through this recipe:

```
# Load libraries
from sklearn.datasets import load_boston
from sklearn.cross_validation import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error
```

```
from sklearn.preprocessing import PolynomialFeatures
import numpy as np
import matplotlib.pyplot as plt

def get_data():
    """
    Return boston dataset
    as x - predictor and
    y - response variable
    """
    data = load_boston()
    x    = data['data']
    y    = data['target']
    return x,y

def build_model(x,y,n_estimators=500):
    """
    Build a Gradient Boost regression model
    """
    model = GradientBoostingRegressor(n_estimators=n_
estimators,verbose=10,\n
                                         subsample=0.7, learning_rate= 0.15,max_depth=3,random_
state=77)
    model.fit(x,y)
    return model

def view_model(model):
    """
    """
    print "\n Training scores"
    print "=====\\n"
    for i,score in enumerate(model.train_score_):
        print "\\tEstimator %d score %0.3f"%(i+1,score)

    plt.cla()
    plt.figure(1)
    plt.plot(range(1,model.estimators_.shape[0]+1),model.train_score_)
    plt.xlabel("Model Sequence")
    plt.ylabel("Model Score")
    plt.show()

    print "\\n Feature Importance"
    print "=====\\n"
```

```
for i,score in enumerate(model.feature_importances_):
    print "\tFeature %d Importance %0.3f"%(i+1,score)

def model_worth(true_y,predicted_y):
    """
    Evaluate the model
    """
    print "\tMean squared error = %0.2f"%(mean_squared_error(true_y,predicted_y))

if __name__ == "__main__":
    x,y = get_data()

    # Divide the data into Train, dev and test
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_size = 0.3,random_state=9)
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_all,test_size=0.3,random_state=9)

    #Prepare some polynomial features
    poly_features = PolynomialFeatures(2,interaction_only=True)
    poly_features.fit(x_train)
    x_train_poly = poly_features.transform(x_train)
    x_dev_poly   = poly_features.transform(x_dev)

    # Build model with polynomial features
    model_poly = build_model(x_train_poly,y_train)
    predicted_y = model_poly.predict(x_train_poly)
    print "\n Model Performance in Training set (Polynomial
features)\n"
    model_worth(y_train,predicted_y)

    # View model details
    view_model(model_poly)

    # Apply the model on dev set
    predicted_y = model_poly.predict(x_dev_poly)
```

```
print "\n Model Performance in Dev set (Polynomial features)\n"
model_worth(y_dev,predicted_y)

# Apply the model on Test set
x_test_poly = poly_features.transform(x_test)
predicted_y = model_poly.predict(x_test_poly)

print "\n Model Performance in Test set (Polynomial features)\n"
model_worth(y_test,predicted_y)
```

Let us write the following three functions.

The function build_model, which implements the Gradient Boosting routine.

The functions view_model and model_worth, which are used to inspect the model that we have built:

```
def build_model(x,y,n_estimators=500):
    """
    Build a Gradient Boost regression model
    """
    model = GradientBoostingRegressor(n_estimators=n_
estimators,verbose=10,\n
        subsample=0.7, learning_rate= 0.15,max_depth=3,random_
state=77)
    model.fit(x,y)
    return model

def view_model(model):
    """
    """
    print "\n Training scores"
    print "=====\\n"
    for i,score in enumerate(model.train_score_):
        print "\tEstimator %d score %.3f"%(i+1,score)

    plt.cla()
    plt.figure(1)
    plt.plot(range(1,model.estimators_.shape[0]+1),model.train_score_)
    plt.xlabel("Model Sequence")
    plt.ylabel("Model Score")
    plt.show()

    print "\n Feature Importance"
    print "=====\\n"
```

```
for i,score in enumerate(model.feature_importances_):
    print "\tFeature %d Importance %0.3f"%(i+1,score)

def model_worth(true_y,predicted_y):
    """
    Evaluate the model
    """
    print "\tMean squared error = %0.2f"%(mean_squared_error(true_y,predicted_y))
```

Finally, we write our main function which will call the other functions:

```
if __name__ == "__main__":
    x,y = get_data()

    # Divide the data into Train, dev and test
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_size = 0.3,random_state=9)
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_all,test_size=0.3,random_state=9)

    #Prepare some polynomial features
    poly_features = PolynomialFeatures(2,interaction_only=True)
    poly_features.fit(x_train)
    x_train_poly = poly_features.transform(x_train)
    x_dev_poly = poly_features.transform(x_dev)

    # Build model with polynomial features
    model_poly = build_model(x_train_poly,y_train)
    predicted_y = model_poly.predict(x_train_poly)
    print "\n Model Performance in Training set (Polynomial features)\n"
    model_worth(y_train,predicted_y)

    # View model details
    view_model(model_poly)

    # Apply the model on dev set
    predicted_y = model_poly.predict(x_dev_poly)
```

```
print "\n Model Performance in Dev set (Polynomial features)\n"
model_worth(y_dev,predicted_y)

# Apply the model on Test set
x_test_poly = poly_features.transform(x_test)
predicted_y = model_poly.predict(x_test_poly)

print "\n Model Performance in Test set (Polynomial features)\n"
model_worth(y_test,predicted_y)
```

How it works...

Let us start with the main module and follow the code. We load the predictor x and response variable y using the get_data function:

```
def get_data():
    """
    Return boston dataset
    as x - predictor and
    y - response variable
    """
    data = load_boston()
    x = data['data']
    y = data['target']
    return x,y
```

The function invokes the Scikit learn's convenience function `load_boston()` to retrieve the Boston house pricing dataset as numpy arrays.

We proceed to divide the data into the train and test sets using the `train_test_split` function from Scikit library. We reserve 30 percent of our dataset for testing.

```
x_train,x_test_all,y_train,y_test_all =
train_test_split(x,y,test_size = 0.3,random_state=9)
```

Out of this 30 percent, we again extract the dev set in the next line:

```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)
```

We proceed to build the polynomial features as follows:

```
poly_features = PolynomialFeatures(interaction_only=True)
poly_features.fit(x_train)
```

As you can see, we have set `interaction_only` to True. By having `interaction_only` set to true, given say `x1` and `x2` attribute, only `x1*x2` attribute is created. Square of `x1` and square of `x2` are not created, assuming that the degree is 2. The default degree is 2.

```
x_train_poly = poly_features.transform(x_train)
x_dev_poly = poly_features.transform(x_dev)
x_test_poly = poly_features.transform(x_test)
```

Using the `transform` function, we transform our train, dev, and test datasets to include polynomial features:

Let us proceed to build our model:

```
# Build model with polynomial features
model_poly = build_model(x_train_poly,y_train)
```

Inside the `build_model` function, we instantiate the `GradientBoostingRegressor` class as follows:

```
model = GradientBoostingRegressor(n_estimators=n_
estimators,verbose=10,\n        subsample=0.7, learning_rate= 0.15,max_depth=3,random_
state=77)
```

Let us look at the parameters. The first parameter is the number of models in the ensemble. The second parameter is `verbose`—when this is set to a number greater than 1, it prints the progress as every model, trees in this case is built. The next parameter is `subsample`, which dictates the percentage of training data that will be used by the models. In this case, 0.7 indicates that we will use 70 percent of the training dataset. The next parameter is the `learning rate`. It's the shrinkage parameter to control the contribution of each tree. `Max_depth`, the next parameter, determines the size of the tree built. The `random_state` parameter is the seed to be used by the random number generator. In order to stay consistent during different runs, we set this to an integer value.

Since we have set our verbose parameter to more than 1, as we fit our model, we see the following results on the screen during each model iteration:

Iter	Train Loss	OOB Improve	Remaining Time
1	58.5196	20.8748	2.49s
2	45.2833	10.3732	1.99s
3	40.1522	8.8467	1.82s
4	27.7772	8.2210	1.86s
5	27.6316	3.9991	1.78s
6	21.0990	4.0621	1.73s
7	17.5833	2.5910	1.76s
8	15.1718	2.3592	1.78s
9	11.9584	2.0957	1.75s
10	10.1687	1.4597	1.72s
11	9.5268	0.8509	1.73s
12	7.2505	0.6745	1.75s
13	6.5691	0.5004	1.76s
14	6.3947	0.2710	1.77s
15	5.9843	0.2344	1.75s
16	5.4427	0.1878	1.78s
17	4.4250	0.4125	1.79s
18	4.4652	0.0001	1.79s
19	4.4287	-0.1490	1.80s
20	4.4158	0.1507	1.80s
21	3.8329	0.0701	1.80s
22	3.9870	-0.0092	1.78s
23	3.4025	-0.1266	1.80s
24	3.7361	-0.0166	1.78s
25	3.5310	0.0589	1.79s

As you can see, the training loss reduces with each iteration. The fourth column is the out-of-bag improvement score. With the subsample, we had selected only 70 percent of the dataset; the OOB score is calculated with the rest 30 percent. There is an improvement in loss as compared to the previous model. For example, in iteration 2, we have an improvement of 10.32 when compared with the model built in iteration 1.

Let us proceed to check performance of the ensemble on the training data:

```
predicted_y = model_poly.predict(x_train_poly)
print "\n Model Performance in Training set (Polynomial
features)\n"
model_worth(y_train,predicted_y)
```

Model Performance in Training set (Polynomial features)

Mean squared error = 0.00

As you can see, our boosting ensemble has fit the training data perfectly.

The model_worth function prints some more details of the model. They are as follows:

```
Training scores
=====
Estimator 1 score 58.520
Estimator 2 score 45.283
Estimator 3 score 40.152
Estimator 4 score 27.777
Estimator 5 score 27.632
Estimator 6 score 21.099
Estimator 7 score 17.583
Estimator 8 score 15.172
Estimator 9 score 11.958
Estimator 10 score 10.169
Estimator 11 score 9.527
Estimator 12 score 7.250
Estimator 13 score 6.569
Estimator 14 score 6.395
Estimator 15 score 5.984
```

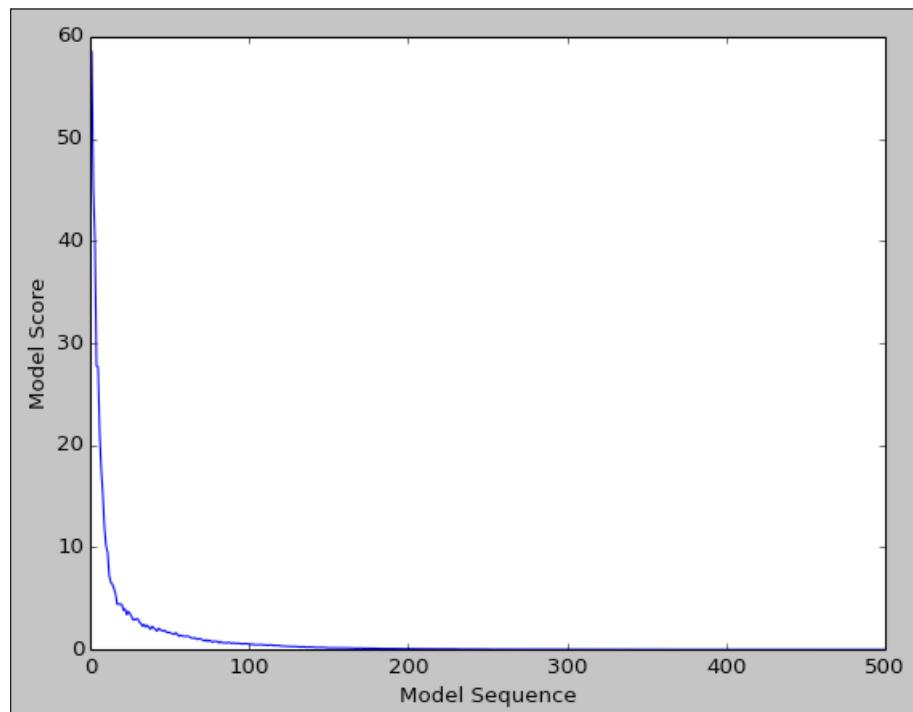
The score of each of the different models, which we saw in the verbose output is stored as an attribute in the model object, and is retrieved as follows:

```
print "\n Training scores"
print "=====\\n"
for i,score in enumerate(model.train_score_):
    print "\\tEstimator %d score %0.3f"%(i+1,score)
```

Let us plot this in a graph:

```
plt.cla()
plt.figure(1)
plt.plot(range(1,model.estimators_.shape[0]+1),model.train_score_)
plt.xlabel("Model Sequence")
plt.ylabel("Model Score")
plt.show()
```

The x axis represents the model number and the y axis displays the training score. Remember that boosting is a sequential process, and every model is an improvement over the previous model.



As you can see in the graph, the mean square error, which is the model score decreases with every successive model.

Finally, we can also see the importance associated with each feature:

```
print "\n Feature Importance"
print "=====\n"
for i,score in enumerate(model.feature_importances_):
    print "\tFeature %d Importance %0.3f"%(i+1,score)
```

Let us see how the features are stacked against each other.

Feature Importance	
=====	
Feature 1	Importance 0.000
Feature 2	Importance 0.004
Feature 3	Importance 0.000
Feature 4	Importance 0.001
Feature 5	Importance 0.000
Feature 6	Importance 0.003
Feature 7	Importance 0.036
Feature 8	Importance 0.012
Feature 9	Importance 0.008
Feature 10	Importance 0.000
Feature 11	Importance 0.005
Feature 12	Importance 0.003
Feature 13	Importance 0.021
Feature 14	Importance 0.010
Feature 15	Importance 0.005
Feature 16	Importance 0.012
Feature 17	Importance 0.001
Feature 18	Importance 0.011
Feature 19	Importance 0.010
Feature 20	Importance 0.011
Feature 21	Importance 0.020
Feature 22	Importance 0.007
Feature 23	Importance 0.005
Feature 24	Importance 0.006
Feature 25	Importance 0.019

Gradient Boosting unifies feature selection and model building into a single operation. It can naturally discover the non-linear relationship between features. Please refer to the following paper on how Gradient boosting can be used for feature selection:

Zhixiang Xu, Gao Huang, Kilian Q. Weinberger, and Alice X. Zheng. 2014. Gradient boosted feature selection. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining(KDD '14). ACM, New York, NY, USA, 522-531.

Let us apply the dev data to the model and look at its performance:

```
# Apply the model on dev set
predicted_y = model_poly.predict(x_dev_poly)
print "\n Model Performance in Dev set  (Polynomial features)\n"
model_worth(y_dev,predicted_y)
```

Model Performance in Dev set (Polynomial features)

Mean squared error = 10.47

Finally, we look at the test set performance.

Model Performance in Test set (Polynomial features)

Mean squared error = 7.15

As you can see, our ensemble has performed extremely well in our test set as compared to the dev set.

There's more...

For more information about Gradient Boosting, please refer to the following paper:

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232.

In this receipt we explained gradient boosting with a squared loss function. However Gradient Boosting should be viewed as a framework and not as a method. Any differentiable loss function can be used in this framework. Any learning method and a differentiable loss function can be chosen by users and apply it into the Gradient Boosting framework.

Scikit Learn also provides a Gradient Boosting method for classification, called GradientBosstingClassifier.

<http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html>

See also

- ▶ *Understanding Ensemble, Bagging Method* recipe in *Chapter 8, Model Selection and Evaluation*
- ▶ *Understanding Ensemble, Boosting Method AdaBoost* recipe in *Chapter 8, Model Selection and Evaluation*
- ▶ *Predicting real valued numbers using regression* recipe in *Chapter 7, Machine Learning II*
- ▶ *Variable Selection using LASSO Regression* recipe in *Chapter 7, Machine Learning II*
- ▶ *Using cross validation iterators* recipe in *Chatper 7, Machine Learning II*

9

Growing Trees

In this chapter, we will cover the following recipes:

- ▶ Going from trees to forest – Random Forest
- ▶ Growing extremely randomized Trees
- ▶ Growing a rotation forest

Introduction

In this chapter, we will see some more Bagging methods based on tree-based algorithms. Due to their robustness against noise and universal applicability to a variety of problems, they are very popular among the data science community.

The claim to fame for most of these methods is that they can obtain very good results with zero data preparation compared to other methods, and they can be provided as black box tools in the hands of software engineers.

Other than the tall claims made in the previous paragraphs, there are some other advantages as well.

By design, bagging lends itself nicely to parallelization. Hence, these methods can be easily applied on a very large dataset in a cluster environment.

Decision Tree algorithms split the input data into various regions at each level of the tree. Thus, they perform implicit feature selection. Feature selection is one of the most important tasks in building a good model. By providing implicit feature selection, Decision Trees are in an advantageous position compared to other techniques. Hence, Bagging with Decision Trees comes with this advantage.

Almost no data preparation is needed for decision trees. For example, consider scaling of attributes. The attribute scale has no impact on the structure of the decision trees. Moreover, missing values do not affect decision trees. The effect of outliers too is very minimal on a Decision Tree.

In some of our earlier recipes, we had used Polynomial features retaining only the interaction components. With an ensemble of trees, these interactions are taken care of. We don't have to make explicit feature transformations to accommodate feature interactions.

Linear Regression-based models fail in the case of the existence of a non-linear relationship in the input data. We saw this effect when we explained the Kernel PCA recipes. Tree-based algorithms are not affected by a non-linear relationship in the data.

One of the major complaints against the Tree-based method is the difficulty with pruning of trees to avoid overfitting. Big trees tend to fit the noise present in the underlying data as well, and hence, lead to a low bias and high variance. However, when we grow a lot of trees, and the final prediction is an average of the output of all the trees in the ensemble, we avoid the problem of variance.

In this chapter, we will see three tree-based ensemble methods.

Our first recipe is about implementing Random Forests for a classification problem. Leo Breiman is the inventor of this algorithm. The Random Forest is an ensemble technique which leverages a lot of trees internally to produce a model for solving any regression or classification problems.

Our second recipe is about Extremely Randomized trees, an algorithm which varies in a very small way from Random Forests. By introducing more randomization in its procedure as compared to a Random Forest, it claims to address the variance problem more effectively. Moreover, it has a slightly reduced computational complexity.

Our final recipe is about Rotation Forests. The first two recipes require a large number of trees to be a part of their ensemble for achieving good performance. Rotation forest claim that they can achieve similar or better performance with a fewer number of trees. Furthermore, the authors of this algorithm claim that the underlying estimator can be anything other than a tree. In this way, it is projected as a new framework for building an ensemble similar to Gradient Boosting.

Going from trees to Forest – Random Forest

The Random forest method builds a lot of trees (forest) which are uncorrelated to each other. Given a classification or a regression problem, the method proceeds to build a lot of trees, and the final prediction is either the average of predictions from the entire forest for regression or a majority vote classification.

This should remind you of Bagging. Random Forests is yet another Bagging methodology. The fundamental idea behind bagging is to use a lot of noisy estimators, handling the noise by averaging, and hence reducing the variance in the final output. Trees are highly affected by even a very small noise in the training dataset. Hence, being a noisy estimator, they are an ideal candidate for Bagging.

Let us write down the steps involved in building a Random Forest. The number of trees required in the forest is a parameter specified by the user. Let T be the number of trees required to be built:

We start with iterating from 1 through T , that is, we build T trees:

- ▶ For each tree, draw a bootstrap sample of size D from our input dataset.
- ▶ We proceed to fit a tree t to the input data:
 - Randomly select m attributes.
 - Pick the best attribute to use as a splitting variable using a predefined criterion.
 - Split the data set into two. Remember, trees are binary in nature. At each level of the tree, the input dataset is split into two.
 - We proceed to do the preceding three steps recursively on the dataset that we split.
- ▶ Finally, we return T trees.

To make a prediction on a new instance, we take a majority vote amongst all the trees in T for a classification; for regression, we take the average value returned by each tree t in T .

We said earlier that a Random Forest builds non-correlated trees. Let's see how the various trees in the ensemble are not correlated to each other. By taking a bootstrap sample from the dataset for each tree, we ensure that different parts of the data are presented to different trees. This way, each tree tries to model different characteristics of the dataset. Hence, we stick to the ensemble rule of introducing variation in the underlying estimators. But this does not guarantee complete non correlation between the underlying trees. When we do the node splitting, we don't select all attributes; rather, we randomly select a subset of attributes. In this manner, we try to ensure that our trees are not correlated to each other.

Compared to Boosting, where our ensemble of estimators were weak classifiers, in a Random Forest, we build trees with maximum depth so that they fit the bootstrapped sample perfectly leading to a low bias. The consequence is the introduction of high variance. However, by building a large number of trees and using the averaging principle for the final prediction, we hope to tackle this variance problem.

Let us proceed to jump into our recipe for a Random Forest.

Getting ready

We are going to generate some classification datasets to demonstrate a Random Forest Algorithm. We will leverage scikit-learn's implementation of a Random Forest from the ensemble module.

How to do it...

We will start with loading all the necessary libraries. Let us leverage the `make_classification` method from the `sklearn.dataset` module for generating the training data to demonstrate a Random Forest:

```
from sklearn.datasets import make_classification
from sklearn.metrics import classification_report, accuracy_score
from sklearn.cross_validation import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.grid_search import RandomizedSearchCV
from operator import itemgetter

import numpy as np

def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 30
    redundant_features = int(0.1*no_features)
    informative_features = int(0.6*no_features)
    repeated_features = int(0.1*no_features)
    x,y = make_classification(n_samples=500,n_features=no_
    features,flip_y=0.03,\n        n_informative = informative_features, n_redundant =
    redundant_features \
        ,n_repeated = repeated_features,random_state=7)
    return x,y
```

We will now write the function `build_forest` to build fully grown trees and proceed to evaluate the forest's performance. Then we will write the methods which can be used to search the optimal parameters for our forest:

```
def build_forest(x,y,x_dev,y_dev):
    """
    Build a random forest of fully grown trees
    and evaluate peformance
```

```
"""
no_trees = 100
estimator = RandomForestClassifier(n_estimators=no_trees)
estimator.fit(x,y)

train_predcited = estimator.predict(x)
train_score = accuracy_score(y,train_predcited)
dev_predicted = estimator.predict(x_dev)
dev_score = accuracy_score(y_dev,dev_predicted)

print "Training Accuracy = %0.2f Dev Accuracy = %0.2f"%(train_
score,dev_score)

def search_parameters(x,y,x_dev,y_dev):
    """
    Search the parameters of random forest algorithm
    """
    estimator = RandomForestClassifier()
    no_features = x.shape[1]
    no_iterations = 20
    sqr_no_features = int(np.sqrt(no_features))

    parameters = {"n_estimators" : np.random.randint(75,200,no_
iterations),
                   "criterion" : ["gini", "entropy"],
                   "max_features" : [sqr_no_features,sqr_no_
features*2,sqr_no_features*3,sqr_no_features+10]
    }

    grid = RandomizedSearchCV(estimator=estimator,param_
distributions=parameters,\
    verbose=1, n_iter=no_iterations,random_state=77,n_jobs=-1,cv=5)
    grid.fit(x,y)
    print_model_worth(grid,x_dev,y_dev)

    return grid.best_estimator_

def print_model_worth(grid,x_dev,y_dev):
    # Print the goodness of the models
    # We take the top 5 models
    scores = sorted(grid.grid_scores_, key=itemgetter(1),
reverse=True) [0:5]
```

```
for model_no,score in enumerate(scores):
    print "Model %d, Score = %0.3f"%(model_no+1,score.mean_
validation_score)
    print "Parameters = {0}".format(score.parameters)
print
dev_predicted = grid.predict(x_dev)

print classification_report(y_dev,dev_predicted)
```

Finally, we write a main function for invoking the functions that we have defined previously:

```
if __name__ == "__main__":
x,y = get_data()

# Divide the data into Train, dev and test
x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)

build_forest(x_train,y_train,x_dev,y_dev)
model = search_parameters(x,y,x_dev,y_dev)
get_feature_importance(model)
```

How it works...

Let us start with our main function. We invoke `get_data` to get our predictor attributes `x` and the response attributes `y`. Inside `get_data`, we leverage the `make_classification` dataset to generate our training data for Random Forest:

```
def get_data():
"""
Make a sample classification dataset
Returns : Independent variable y, dependent variable x
"""
no_features = 30
redundant_features = int(0.1*no_features)
informative_features = int(0.6*no_features)
repeated_features = int(0.1*no_features)
x,y = make_classification(n_samples=500,n_features=no_
features,flip_y=0.03,\n_informative = informative_features, n_redundant =
redundant_features \
,n_repeated = repeated_features,random_state=7)
return x,y
```

Let us look at the parameters passed to the `make_classification` method. The first parameter is the number of instances required; in this case, we say we need 500 instances. The second parameter is about the number of attributes required per instance. We say that we need 30. The third parameter, `flip_y`, randomly interchanges 3 percent of the instances. This is done to introduce some noise in to our data. The next parameter specifies the number of features out of those 30 features, which should be informative enough to be used in our classification. We have specified that 60 percent of our features, that is, 18 out of 30 should be informative. The next parameter is about the redundant features. These are generated as a linear combination of the informative features in order to introduce a correlation among the features. Finally, repeated features are the duplicate features, which are drawn randomly from both informative features and redundant features.

Let us split the data into the training and testing set using `train_test_split`. We reserve 30 percent of our data for testing:

```
# Divide the data into Train, dev and test
x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
```

Once again, we leverage `train_test_split` to split our test data into dev and test:

```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_-
all,test_size=0.3,random_state=9)
```

With the data divided for building, evaluating, and testing the model, we proceed to build our models:

```
build_forest(x_train,y_train,x_dev,y_dev)
```

We invoke the `build_forest` function with our training and dev data to build the random forest model. Let us look inside that function:

```
no_trees = 100
estimator = RandomForestClassifier(n_estimators=no_trees)
estimator.fit(x,y)

train_predcited = estimator.predict(x)
train_score = accuracy_score(y,train_predcited)
dev_predicted = estimator.predict(x_dev)
dev_score = accuracy_score(y_dev,dev_predicted)

print "Training Accuracy = %0.2f Dev Accuracy = %0.2f"%(train_
score,dev_score)
```

We need 100 trees in our ensemble, so we use the variable `no_trees` to define the number of trees. We leverage the `RandomForestClassifier` class from scikit-learn check and apply throughout. As you can see, we pass the number of trees required as a parameter. We then proceed to fit our model.

Now let us find the model accuracy score for our train and dev data:

```
Training Accuracy = 1.00 Dev Accuracy = 0.83
```

Not bad! We have achieved 83 percent accuracy on our dev set. Let us see if we can improve our scores. There are other parameters to the forest which can be tuned to get a better model. For the list of parameters which can be tuned, refer to the following link:

<http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

We invoke the function `search_parameters` with the training and dev data to tune the various parameters for our Random Forest model.

In some of the previous recipes, we used `GridSearchCV` to search through the parameter space for finding the best parameter combination. `GridSearchCV` performs a very exhaustive search. However, in this recipe we are going to use `RandomizedSearchCV`. We provide a distribution of parameter values for each parameter, and specify the number of iterations needed. For each iteration, `RandomizedSearchCV` will pick a sample value from the parameter distribution and fit the model:

```
parameters = {"n_estimators" : np.random.randint(75,200,no_iterations),
              "criterion" : ["gini", "entropy"],
              "max_features" : [sqr_no_features,sqr_no_features*2,sqr_no_features*3,sqr_no_features+10]
}
```

We provide a dictionary of parameters as we did in `GridSearchCV`. In our case, we want to experiment with three parameters.

The first one is the number of trees in the model, represented by the `n_estimators` parameter. By invoking the `randint` function, we get a list of integers between 75 and 200. The size of the trees is defined by `no_iterations` parameters:

```
no_iterations = 20
```

This is the parameter we will pass to `RandomizedSearchCV` for the number of iterations we want to perform. From this array of 20 elements, `RandomizedSearchCV` will sample a single value for each iteration.

Our next parameter is the criterion, we pick randomly between `gini` and `entropy`, and use that as a criterion for splitting the nodes during each iteration.

The most important parameter, `max_features`, defines the number of features that the algorithm should pick during the splitting of each node. In our pseudocode for describing the Random Forest, we have specified that we need to pick m attributes randomly during each split of the node. The parameter `max_features` defines that m . Here we give a list of four values. The variable `sqr_no_features` is the square root of the number of attributes available in the input dataset:

```
sqr_no_features = int(np.sqrt(no_features))
```

Other values in that list are some variations of the square root.

Let us instantiate `RandomizedSearchCV` with this parameter distribution:

```
grid = RandomizedSearchCV(estimator=estimator, param_distributions=parameters,\\
                           verbose=1, n_iter=no_iterations, random_state=77, n_jobs=-1, cv=5)
```

The first parameter is the underlying estimator whose parameters we are trying to optimize. It's our `RandomForestClassifier`:

```
estimator = RandomForestClassifier()
```

The second parameter, `param_distributions` is the distribution defined by the dictionary `parameters`. We define the number of iterations, that is, the number of times we want to run the `RandomForestClassifier` using the parameter `n_iter`. With the `cv` parameter, we specify the number of cross validations required 5 cross validations in our case.

Let us proceed to fit the model, and see how well the model has turned out:

```
grid.fit(x,y)
print_model_worth(grid,x_dev,y_dev)
```

Fitting 5 folds for each of 20 candidates, totalling 100 fits

As you can see, we have five folds, that is, we want to do a five-fold cross validation on each of our iterations. We have a total of 20 iterations, and hence, we will be building 100 models.

Let us look inside the function `print_model_worth`. We pass our `grid` object and `dev` dataset to this function. The `grid` object stores the evaluation metric for each of the models it builds inside an attribute called the `grid_scores_` of type list. Let us sort this list in the descending order to build the best model:

```
scores = sorted(grid.grid_scores_, key=itemgetter(1), reverse=True)[0:5]
```

We select the top five models as you can see from the indexing. We proceed to print the details of those models:

```
for model_no,score in enumerate(scores):
    print "Model %d, Score = %0.3f"%(model_no+1,score.mean_validation_
    score)
    print "Parameters = {0}".format(score.parameters)
    print
```

We first print the evaluation score and follow it with the parameters of the model:

```
Model 1, Score = 0.864
Parameters = {'max_features': 5, 'n_estimators': 197, 'criterion':
'entropy'}
Model 2, Score = 0.856
Parameters = {'max_features': 5, 'n_estimators': 182, 'criterion':
'gini'}
Model 3, Score = 0.848
Parameters = {'max_features': 15, 'n_estimators': 118, 'criterion':
'gini'}
Model 4, Score = 0.846
Parameters = {'max_features': 5, 'n_estimators': 186, 'criterion':
'entropy'}
Model 5, Score = 0.846
Parameters = {'max_features': 5, 'n_estimators': 177, 'criterion':
'gini'}
```

We have ranked the modes by their scores in the descending order thus showing the best model parameters in the beginning. We will choose these parameters as our model parameters. The attribute `best_estimator_` will return the model with these parameters.

Let us use these parameters and test our dev data:

```
dev_predicted = grid.predict(x_dev)
print classification_report(y_dev,dev_predicted)
```

The predict function will use `best_estimator_` internally:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	51
1	1.00	1.00	1.00	54
avg / total	1.00	1.00	1.00	105

Great! We have a perfect model with a classification accuracy of 100 percent.

There's more...

Internally, the `RandomForestClassifier` uses the `DecisionTreeClassifier`. Refer to the following link for all the parameters that are passed for building a decision tree:

<http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>

One parameter that is of some interest to us is `splitter`. The default value of `splitter` is set to `best`. Based on the `max_features` attribute, the implementation will choose the splitting mechanism internally. The available splitting mechanisms include the following:

- ▶ `best`: Chooses the best possible split from the given set of attributes defined by the `max_features` parameter
- ▶ `random`: Randomly chooses a splitting attribute

You would have noticed that this parameter is not available while instantiating a `RandomForestClassifier`. The only way to control is to give a value to the `max_features` parameter which is less than the number of attributes available in the dataset.

In the industry, Random Forests are extensively used for variable selection. In Scikit learn, variable importance is calculated using gini impurity. Both the gini and entropy criteria used for node splitting identify the best attribute for splitting the node by its ability to split the dataset into subsets with high impurity so that subsequent splitting leads to good classification. The importance of a variable is decided by the amount of impurity it can induce into the split dataset. Refer to the following book for more details:

Breiman, Friedman, "Classification and regression trees", 1984.

We can write a small function to print the important features:

```
def get_feature_importance(model):  
    feature_importance = model.feature_importances_  
    fm_with_id = [(i,importance) for i,importance in  
    enumerate(feature_importance)]  
    fm_with_id = sorted(fm_with_id, key=itemgetter(1),reverse=True)  
    [0:10]  
    print "Top 10 Features"  
    for importance in fm_with_id:  
        print "Feature %d importance = %0.3f"%(importance[0],importan  
ce[1])  
    print
```

A Random Forest object has a variable called `feature_importances_`. We use this variable and create a list of tuples with the feature number and importance:

```
feature_importance = model.feature_importances_  
fm_with_id = [(i,importance) for i,importance in  
enumerate(feature_importance)]
```

We proceed to sort it in descending order of importance, and select only the top 10 features:

```
fm_with_id = sorted(fm_with_id, key=itemgetter(1), reverse=True)  
[0:10]
```

We then print the top 10 features:

```
Top 10 Features  
Feature 16 importance = 0.086  
Feature 17 importance = 0.074  
Feature 19 importance = 0.050  
Feature 24 importance = 0.049  
Feature 27 importance = 0.049  
Feature 10 importance = 0.045  
Feature 0 importance = 0.042  
Feature 3 importance = 0.040  
Feature 5 importance = 0.038  
Feature 11 importance = 0.038
```

Another interesting aspect of Random Forests is the Out-of-Bag estimation (OOB). Remember that we bootstrap from the dataset initially for every tree grown in the forest. Because of bootstrapping, some records will not be used in some trees. Let us say record 1 is used in 100 trees and not used in 150 trees in our forest. We can then use those 150 trees to predict the class label for that record to figure out the classification error for that record. Out-of-bag estimation can be used to effectively assess the quality of our forest. The following URL gives an example of how the OOB can be used effectively:

http://scikit-learn.org/dev/auto_examples/ensemble/plot_ensemble_oob.html

The RandomForestClassifier class in Scikit learn is derived from ForestClassifier. The source code for the same can be found at the following link:

<https://github.com/scikit-learn/scikit-learn/blob/a95203b/sklearn/ensemble/forest.py#L318>

When we call the predict method in RandomForestClassifier, it internally calls the predict_proba method defined in ForestClassifier. Here, the final prediction is done not on the basis of voting but by averaging the probabilities for each of the classes from different trees inside the forest and deciding the final class based on the highest probability.

The original paper by Leo Breiman on Random Forests is available for download at the following link:

<http://link.springer.com/article/10.1023%2FA%3A1010933404324>

You can also refer to the website maintained by Leo Breiman and Adele Cutler:

https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

See also

- ▶ *Building Decision Trees to solve Multi Class Problems* recipe in Chapter 6, Machine Learning I
- ▶ *Understanding Ensemble, Gradient Boosting* recipe in Chapter 8, Model Selection and Evaluation
- ▶ *Understanding Ensemble, Bagging Method* recipe in Chapter 8, Model Selection and Evaluation

Growing Extremely Randomized Trees

Extremely Randomized Trees, also known as the Extra trees algorithm differs from the Random Forest described in the previous recipe in two ways:

1. It does not use bootstrapping to select instances for every tree in the ensemble; instead, it uses the complete training dataset.
2. Given K as the number of attributes to be randomly selected at a given node, it selects a random cut-point without considering the target variable.

As you saw in the previous recipe, Random Forests used randomization in two places. First, in selecting the instances to be used for the training trees in the forest; bootstrap was used to select the training instances. Secondly, at every node a random set of attributes were selected. One attribute among them was selected based on either the gini impurity or entropy criterion. Extremely randomized trees go one step further and select the splitting attribute randomly.

Extremely Randomized Trees were proposed in the following paper:

P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", *Machine Learning*, 63(1), 3-42, 2006.

According to this paper, there are two aspects, other than the technical aspects listed earlier, which make an Extremely Randomized Tree more suitable:

The rationale behind the Extra-Trees method is that the explicit randomization of the cut-point and attribute combined with ensemble averaging should be able to reduce variance more strongly than the weaker randomization schemes used by other methods.

Compared to a Random Forest, randomization of the cut-point (the attribute selected to split the dataset at each node) combined with the randomization of cut-point, that is, ignoring any criteria, and finally, averaging the results from each of the tree, will result in a much superior performance on an unknown dataset.

The second advantage is regarding the compute complexity:

From the computational point of view, the complexity of the tree growing procedure is, assuming balanced trees, on the order of $N \log N$ with respect to learning the sample size, like most other tree growing procedures. However, given the simplicity of the node splitting procedure we expect the constant factor to be much smaller than in other ensemble based methods which locally optimize cut-points

Since no computation time is spent in identifying the best attribute to split, this method is more computationally efficient than Random Forests.

Let us write down the steps involved in building Extremely Random trees. The number of trees required in the forest is typically specified by the user. Let T be the number of trees required to be built.

We start with iterating from 1 through T , that is, we build T trees:

- ▶ For each tree, we select the complete input dataset.
- ▶ We then proceed to fit a tree t to the input data:
 - Select m attributes randomly.
 - Pick an attribute randomly as the splitting variable.
 - Split the data set into two. Remember that trees are binary in nature. At each level of the tree, the input dataset is split into two.
 - Perform the preceding three steps recursively on the dataset that we split.
- ▶ Finally, we return T trees.
- ▶ Let us take a look at the recipe for Extremely Randomized Trees.

Getting ready...

We are going to generate some classification datasets to demonstrate Extremely Randomized Trees. For that, we will leverage Scikit Learn's implementation of the Extremely Randomized Trees ensemble module.

How to do it...

We start by loading all the necessary libraries. Let us leverage the `make_classification` method from the `sklearn.dataset` module to generate the training data:

```
from sklearn.datasets import make_classification  
from sklearn.metrics import classification_report, accuracy_score
```

```
from sklearn.cross_validation import train_test_split, cross_val_
score
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.grid_search import RandomizedSearchCV
from operator import itemgetter

def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 30
    redundant_features = int(0.1*no_features)
    informative_features = int(0.6*no_features)
    repeated_features = int(0.1*no_features)
    x,y = make_classification(n_samples=500,n_features=no_
features,flip_y=0.03,\
        n_informative = informative_features, n_redundant =
redundant_features \
        ,n_repeated = repeated_features,random_state=7)
    return x,y
```

We write the function `build_forest`, where we will build fully grown trees, and proceed to evaluate the forest's performance:

```
def build_forest(x,y,x_dev,y_dev):
    """
    Build a Extremely random tress
    and evaluate peformance
    """
    no_trees = 100
    estimator = ExtraTreesClassifier(n_estimators=no_trees,random_
state=51)
    estimator.fit(x,y)

    train_predcited = estimator.predict(x)
    train_score = accuracy_score(y,train_predcited)
    dev_predicted = estimator.predict(x_dev)
    dev_score = accuracy_score(y_dev,dev_predicted)

    print "Training Accuracy = %0.2f Dev Accuracy = %0.2f"%(train_
score,dev_score)
    print "cross validated score"
```

```
print cross_val_score(estimator,x_dev,y_dev,cv=5)

def search_parameters(x,y,x_dev,y_dev):
    """
    Search the parameters
    """
    estimator = ExtraTreesClassifier()
    no_features = x.shape[1]
    no_iterations = 20
    sqr_no_features = int(np.sqrt(no_features))

    parameters = {"n_estimators" : np.random.randint(75,200,no_
iterations),
                   "criterion" : ["gini", "entropy"],
                   "max_features" : [sqr_no_features,sqr_no_
features*2,sqr_no_features*3,sqr_no_features+10]
    }

    grid = RandomizedSearchCV(estimator=estimator,param_
distributions=parameters,\n        verbose=1, n_iter=no_iterations,random_state=77,n_jobs=-1,cv=5)
    grid.fit(x,y)
    print_model_worth(grid,x_dev,y_dev)

    return grid.best_estimator_
```

Finally, we write a main function for invoking the functions that we have defined:

```
if __name__ == "__main__":
    x,y = get_data()

    # Divide the data into Train, dev and test
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)

    build_forest(x_train,y_train,x_dev,y_dev)
    model = search_parameters(x,y,x_dev,y_dev)
```

How it works...

Let us start with our main function. We invoke `get_data` to get our predictor attributes in the response attributes. Inside `get_data`, we leverage the `make_classification` dataset to generate the training data for our recipe as follows:

```
def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 30
    redundant_features = int(0.1*no_features)
    informative_features = int(0.6*no_features)
    repeated_features = int(0.1*no_features)
    x,y = make_classification(n_samples=500,n_features=no_
    features,flip_y=0.03,\n        n_informative = informative_features, n_redundant =
    redundant_features \
        ,n_repeated = repeated_features,random_state=7)
    return x,y
```

Let us look at the parameters that are passed to the `make_classification` method. The first parameter is the number of instances required; in this case, we say we need 500 instances. The second parameter is about the number of attributes required per instance. We say that we need 30. The third parameter, `flip_y`, randomly interchanges 3 percent of the instances. This is done to introduce some noise in our data. The next parameter specifies the number of features out of those 30 features should be informative enough to be used in our classification. We have specified that 60 percent of our features, that is, 18 out of 30 should be informative. The next parameter is about redundant features. These are generated as a linear combination of the informative features in order to introduce a correlation among the features. Finally, repeated features are the duplicate features, which are drawn randomly from both informative features and redundant features.

Let us split the data into the training and testing set using `train_test_split`. We reserve 30 percent of our data for testing:

```
# Divide the data into Train, dev and test
x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
```

Once again, we leverage `train_test_split` to split our test data into dev and test:

```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)
```

With the data divided for building, evaluating, and testing the model, we proceed to build our models:

```
build_forest(x_train,y_train,x_dev,y_dev)
```

We invoke the `build_forest` function with our training and dev data to build our Extremely Randomized trees model. Let us look inside that function:

```
no_trees = 100
estimator = ExtraTreesClassifier(n_estimators=no_trees,random_
state=51)
estimator.fit(x,y)

train_predcited = estimator.predict(x)
train_score = accuracy_score(y,train_predcited)
dev_predicted = estimator.predict(x_dev)
dev_score = accuracy_score(y_dev,dev_predicted)

print "Training Accuracy = %0.2f Dev Accuracy = %0.2f"%(train_
score,dev_score)
print "cross validated score"
print cross_val_score(estimator,x_dev,y_dev,cv=5)
```

We need 100 trees in our ensemble, so we use the variable `no_trees` to define the number of trees. We leverage the `ExtraTreesClassifier` class from Scikit learn. As you can see, we pass the number of trees required as a parameter. A point to note here is the parameter `bootstrap`. Refer to the following URL for the parameters for the `ExtraTreesClassifier`:

<http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>

The parameter `bootstrap` is set to `False` by default. Compare it with the `RandomForestClassifier` `bootstrap` parameter given at the following URL:

<http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

As explained earlier, every tree in the forest is trained with all the records.

We proceed to fit our model as follows:

```
train_predcited = estimator.predict(x)
```

We then proceed to find the model accuracy score for our train and dev data:

```
train_score = accuracy_score(y,train_predcited)
dev_predicted = estimator.predict(x_dev)
dev_score = accuracy_score(y_dev,dev_predicted)
```

Let us print the scores for the training and dev dataset:

```
print "Training Accuracy = %0.2f Dev Accuracy = %0.2f"%(train_
score,dev_score)
```

```
Training Accuracy = 1.00 Dev Accuracy = 0.84
```

Let us now do a five-fold cross validation to look at the model predictions:

```
cross validated score
[ 0.81818182  0.76190476  0.80952381  0.85714286  0.9 ]
```

Pretty good results. We almost have a 90 percent accuracy rate for one of the folds. We can do a randomized search across the parameter space as we did for Random Forest. Let us invoke the function `search_parameters` with our train and test dataset. Refer to the previous recipe for an explanation of `RandomizedSearchCV`. We will then print the output of the `search_parameters` function:

```
Model 1, Score = 0.878
Parameters = {'max_features': 15, 'n_estimators': 123, 'criterion':
'entropy'}
Model 2, Score = 0.876
Parameters = {'max_features': 15, 'n_estimators': 77, 'criterion':
'entropy'}
Model 3, Score = 0.874
Parameters = {'max_features': 15, 'n_estimators': 195, 'criterion':
'gini'}
Model 4, Score = 0.874
Parameters = {'max_features': 10, 'n_estimators': 195, 'criterion':
'gini'}
Model 5, Score = 0.874
Parameters = {'max_features': 15, 'n_estimators': 127, 'criterion':
'gini'}

precision      recall   f1-score   support
          0       1.00      1.00      1.00       51
          1       1.00      1.00      1.00       54

avg / total    1.00      1.00      1.00      105
```

As in the previous recipe, we have ranked the models by their scores in descending order, thus showing the best model parameters in the beginning. We will choose these parameters as our model parameters. The attribute `best_estimator_` will return the model with these parameters.

What you see next is the classification report generated for the best estimator. The predict function will use `best_estimator_` internally. The report was generated by the following code:

```
dev_predicted = grid.predict(x_dev)
print classification_report(y_dev, dev_predicted)
```

Great! We have a perfect model with a classification accuracy of 100 percent.

There's more...

Extremely Randomized Trees are very popular with the time series classification problems. Refer to the following paper for more information:

Geurts, P., Blanco Cuesta A., and Wehenkel, L. (2005a). Segment and combine approach for biological sequence classification. In: Proceedings of IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology, 194–201.

See also

- ▶ *Building Decision Trees to solve Multi Class Problems* recipe in Chapter 6, Machine Learning I
- ▶ *Understanding Ensemble, Bagging Method* recipe in Chapter 8, Model Selection and Evaluation
- ▶ *Growing from trees to Forest, Random Forest* recipe in Chapter 9, Machine Learning III

Growing Rotational Forest

Random forests and Bagging give impressive results with very large ensembles; having a large number of estimators results in an improvement in the accuracy of these methods. On the contrary, a Rotational forest is designed to work with a smaller number of ensembles.

Let us write down the steps involved in building a Rotational Forest. The number of trees required in the forest is typically specified by the user. Let T be the number of trees required to be built.

We start with iterating from 1 through T , that is, we build T trees.

For each tree t , perform the following steps:

- ▶ Split the attributes in the training set into K non-overlapping subsets of equal size.
- ▶ We have K datasets, each with K attributes. For each of the K datasets, we proceed to do the following: Bootstrap 75 percent of the data from each K dataset, and use the bootstrapped sample for further steps:

- Run a Principal Component analysis on the i th subset in K . Retain all the principal components. For every feature j in the K th subset, we have a principal component a . Let us denote it as aij , which is the principal component for the j th attribute in the i th subset.
 - Store the principal components for the subset.
- ▶ Create a rotation matrix of size $n \times n$, where n is the total number of attributes. Arrange the principal components in the matrix such that the components match the position of the features in the original training dataset.
 - ▶ Project the training dataset on the Rotation matrix using matrix multiplication.
 - ▶ Build a decision tree with the projected dataset.
 - ▶ Store the tree and the rotational matrix.

With this knowledge, let us jump to our recipe.

Getting ready...

We are going to generate some classification datasets to demonstrate a Rotational Forest. To our knowledge, there is no Python implementation available for Rotational forests. Hence, we will write our own code. We will leverage Scikit Learn's implementation of a Decision Tree Classifier and use the `train_test_split` method for bootstrapping.

How to do it...

We will start with loading all the necessary libraries. Let us leverage the `make_classification` method from the `sklearn.dataset` module to generate the training data. We follow it with a method to select a random subset of attributes called `gen_random_subset`:

```
from sklearn.datasets import make_classification
from sklearn.metrics import classification_report
from sklearn.cross_validation import train_test_split
from sklearn.decomposition import PCA
from sklearn.tree import DecisionTreeClassifier
import numpy as np

def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 50
    redundant_features = int(0.1*no_features)
    informative_features = int(0.6*no_features)
```

```
repeated_features = int(0.1*no_features)
x,y = make_classification(n_samples=500,n_features=no_
features,flip_y=0.03,\n    n_informative = informative_features, n_redundant =
redundant_features \
    ,n_repeated = repeated_features,random_state=7)
return x,y

def get_random_subset(iterable,k):
    subsets = []
    iteration = 0
    np.random.shuffle(iterable)
    subset = 0
    limit = len(iterable)/k
    while iteration < limit:
        if k <= len(iterable):
            subset = k
        else:
            subset = len(iterable)
        subsets.append(iterable[-subset:])
        del iterable[-subset:]
        iteration+=1
    return subsets
```

We now write a function `build_rotationtree_model`, where we will build fully grown trees, and proceed to evaluate the forest's performance using the function `model_worth`:

```
def build_rotationtree_model(x_train,y_train,d,k):
    models = []
    r_matrices = []
    feature_subsets = []
    for i in range(d):
        x_,_,_,_ = train_test_split(x_train,y_train,test_
size=0.3,random_state=7)
        # Features ids
        feature_index = range(x_.shape[1])
        # Get subsets of features
        random_k_subset = get_random_subset(feature_index,k)
        feature_subsets.append(random_k_subset)
        # Rotation matrix
        R_matrix = np.zeros((x_.shape[1],x_.shape[1]),dtype=float)
        for each_subset in random_k_subset:
            pca = PCA()
            x_subset = x_[:,each_subset]
            pca.fit(x_subset)
```

```
for ii in range(0,len(pca.components_)):
    for jj in range(0,len(pca.components_)):
        R_matrix[each_subset[ii],each_subset[jj]] = pca.
components_[ii,jj]

x_transformed = x_train.dot(R_matrix)

model = DecisionTreeClassifier()
model.fit(x_transformed,y_train)
models.append(model)
r_matrices.append(R_matrix)
return models,r_matrices,feature_subsets

def model_worth(models,r_matrices,x,y):

predicted_ys = []
for i,model in enumerate(models):
    x_mod = x.dot(r_matrices[i])
    predicted_y = model.predict(x_mod)
    predicted_ys.append(predicted_y)

predicted_matrix = np.asmatrix(predicted_ys)
final_prediction = []
for i in range(len(y)):
    pred_from_all_models = np.ravel(predicted_matrix[:,i])
    non_zero_pred = np.nonzero(pred_from_all_models)[0]
    is_one = len(non_zero_pred) > len(models)/2
    final_prediction.append(is_one)

print classification_report(y, final_prediction)
```

Finally, we write a main function for invoking the functions that we have defined earlier:

```
if __name__ == "__main__":
    x,y = get_data()
    # plot_data(x,y)

    # Divide the data into Train, dev and test
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)

    # Build a bag of models
```

```
models,r_matrices,features = build_rotationtree_model(x_train,y_train,25,5)
model_worth(models,r_matrices,x_train,y_train)
model_worth(models,r_matrices,x_dev,y_dev)
```

How it works...

Let us start with our main function. We invoke `get_data` to get our predictor attributes in the response attributes. Inside `get_data`, we leverage the `make_classification` dataset to generate the training data for our recipe as follows:

```
def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 30
    redundant_features = int(0.1*no_features)
    informative_features = int(0.6*no_features)
    repeated_features = int(0.1*no_features)
    x,y = make_classification(n_samples=500,n_features=no_
    features,flip_y=0.03,\
        n_informative = informative_features, n_redundant =
    redundant_features \
        ,n_repeated = repeated_features,random_state=7)
    return x,y
```

Let us look at the parameters passed to the `make_classification` method. The first parameter is the number of instances required; in this case we say we need 500 instances. The second parameter is about the number of attributes required per instance. We say that we need 30. The third parameter, `flip_y`, randomly interchanges 3 percent of the instances. This is done to introduce some noise in our data. The next parameter is about the number of features out of those 30 features, which should be informative enough to be used in our classification. We have specified that 60 percent of our features, that is, 18 out of 30 should be informative. The next parameter is about redundant features. These are generated as a linear combination of the informative features in order to introduce a correlation among the features. Finally, repeated features are the duplicate features which are drawn randomly from both informative features and redundant features.

Let us split the data into the training and testing set using `train_test_split`. We reserve 30 percent of our data for testing:

```
# Divide the data into Train, dev and test
x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
```

Once again, we leverage `train_test_split` to split our test data into dev and test as follows:

```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_all,test_size=0.3,random_state=9)
```

With the data divided for building, evaluating, and testing the model, we proceed to build our models:

```
models,r_matrices,features = build_rotationtree_model(x_train,y_train,25,5)
```

We invoke the `build_rotationtree_model` function to build our Rotational forest. We pass our training data, predictors `x_train` and response variable `y_train`, the total number of trees to be built (25 in this case), and finally, the subset of features to be used (5 in this case).

Let us jump to that function:

```
models = []
r_matrices = []
feature_subsets = []
```

We begin with declaring three lists to store each of the decision tree, the rotation matrix for that tree, and finally, the subset of features used in that iteration. We proceed to build each tree in our ensemble.

As a first order of business, we bootstrap to retain only 75 percent of the data:

```
x_,_,_,_ = train_test_split(x_train,y_train,test_size=0.3,random_state=7)
```

We leverage the `train_test_split` function from Scikit learn for bootstrapping. We then decide the feature subsets as follows:

```
# Features ids
feature_index = range(x.shape[1])
# Get subsets of features
random_k_subset = get_random_subset(feature_index,k)
feature_subsets.append(random_k_subset)
```

The function `get_random_subset` takes the feature index and the number of subsets that require k as parameter, and returns K subsets.

Inside that function, we shuffle the feature index. The feature index is an array of numbers that starts from 0 and ends with the number of features in our training set:

```
np.random.shuffle(iterable)
```

Let us say we have 10 features and our k value is 5 indicating that we need subsets with 5 non-overlapping feature indices; we need to then do two iterations. We store the number of iterations needed in the limit variable:

```
limit = len(iterable) /k
while iteration < limit:
    if k <= len(iterable):
        subset = k
    else:
        subset = len(iterable)
    iteration+=1
```

If our required subset is less than the total number of attributes, then we can proceed to use the first k entries in our iterable. Since we have shuffled our iterables, we will be returning different volumes at different times:

```
subsets.append(iterable[-subset:])
```

On selecting a subset, we remove it from the iterable as we need non-overlapping sets:

```
del iterable[-subset:]
```

With all the subsets ready, we declare our rotation matrix as follows:

```
# Rotation matrix
R_matrix = np.zeros((x.shape[1],x.shape[1]),dtype=float)
```

As you can see, our rotational matrix is of size n x n, where n is the number of attributes in our dataset. You can see that we have used the shape attribute to declare this matrix filled with zeros:

```
for each_subset in random_k_subset:
    pca = PCA()
    x_subset = x[:,each_subset]
    pca.fit(x_subset)
```

For each of the K subsets of data having only K features, we proceed to perform the principal component analysis.

We fill our rotational matrix with the component values as follows:

```
for ii in range(0,len(pca.components_)):
    for jj in range(0,len(pca.components_)):
        R_matrix[each_subset[ii],each_subset[jj]] = pca.
components_[ii,jj]
```

For example, let us say that we have three attributes in our subset, in a total of six attributes. For illustration, let us say our subsets are:

2, 4, 6 and 1, 3, 5

Our rotational matrix R is of size 6 x 6. Assume that we want to fill the rotation matrix for the first subset of features. We will have three principal components, one each for 2, 4, and 6 of size 1 x 3.

The output of the PCA from Scikit learn is a matrix of the size component's X features. We go through each component value in the for loop. At the first run, our feature of interest is 2, and the cell (0,0) in the component matrix output from PCA gives the value of the contribution of feature 2 to component 1. We have to find the right place in the rotational matrix for this value. We use the index from the component matrix ii and jj with the subset list to get the right place in the rotation matrix:

```
R_matrix[each_subset[ii],each_subset[jj]] = pca.components_[ii,jj]
```

each_subset[0] and each_subset[0] will put us in cell (2,2) in the rotation matrix. As we go through the loop, the next component value in cell (0,1) in the component matrix will be placed in cell (2,4) of the rotational matrix, and the last one in cell (2,6) of the rotational matrix. This is done for all the attributes in the first subset. Let us go to the second subset; here the first attribute is 1. Cell (0,0) of the component matrix corresponds to cell (1,1) in the rotation matrix.

Proceeding this way, you will notice that the attribute component values are arranged in the same order as the attributes themselves.

With our rotation matrix ready, let us project our input onto the rotation matrix:

```
x_transformed = x_train.dot(R_matrix)
```

It's time now to fit our decision tree:

```
model = DecisionTreeClassifier()
model.fit(x_transformed,y_train)
```

Finally, we store our models and the corresponding rotation matrices:

```
models.append(model)
r_matrices.append(R_matrix)
```

With our model built, let us proceed to see how good our model is with both the train and the dev data, using the model_worth function:

```
model_worth(models,r_matrices,x_train,y_train)
model_worth(models,r_matrices,x_dev,y_dev)
```

Let us take a look at our model_worth function:

```
for i, model in enumerate(models):
    x_mod = x.dot(r_matrices[i])
    predicted_y = model.predict(x_mod)
    predicted_ys.append(predicted_y)
```

Inside the function we perform prediction using each of the trees we have built. However, before the prediction, we project our input using the rotation matrix. We store all our prediction output in a list called `predicted_ys`. Let us say we have 100 instances to predict, and we have 10 models in our tree. For each instance, we have 10 predictions. We store those as a matrix for convenience:

```
predicted_matrix = np.asmatrix(predicted_ys)
```

Now we proceed to give a final classification for each of our input records:

```
final_prediction = []
for i in range(len(y)):
    pred_from_all_models = np.ravel(predicted_matrix[:,i])
    non_zero_pred = np.nonzero(pred_from_all_models)[0]
    is_one = len(non_zero_pred) > len(models)/2
    final_prediction.append(is_one)
```

We will store our final prediction in a list called `final_prediction`. We go through each of the predictions for our instance. Say we are in the first instance (`i=0` in our for loop); `pred_from_all_models` stores the output from all the trees in our model. It's an array of 0s and 1s indicating the class which has the model classified at that instance.

We make another array out of it `non_zero_pred` which has only those entries from the parent arrays which are non-zero.

Finally, if the length of this non-zero array is greater than half the number of models that we have, we say our final prediction is 1 for the instance of interest. What we have accomplished here is the classic voting scheme.

Let us look at how good our models are now by calling `classification_report`:

```
print classification_report(y, final_prediction)
```

The following is the performance of our model on the training set:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	173
1	1.00	1.00	1.00	177
avg / total	1.00	1.00	1.00	350

Let us look at our model's performance on the dev dataset:

	precision	recall	f1-score	support
0	0.87	0.83	0.85	54
1	0.83	0.86	0.85	51
avg / total	0.85	0.85	0.85	105

There's more...

More information about Rotational forests can be gathered from the following paper:

Rotation Forest: A New Classifier Ensemble Method, Juan J. Rodriguez, Member, IEEE Computer Society, Ludmila I. Kuncheva, Member, IEEE, and Carlos J. Alonso

The paper also claims that when Extremely Randomized Trees was compared to Bagging, AdBoost, and Random Forest on 33 datasets, Extremely Randomized Trees outperformed all the other three algorithms.

Similar to Gradient Boosting, the authors of the paper claim that the Extremely Randomized method is an overall framework, and the underlying ensemble does not necessarily have to be a Decision Tree. Work is in progress on testing other algorithms like Naïve Bayes, Neural Networks, and others.

See also

- ▶ *Extracting Principal Components* recipe in Chapter 4, Analyzing Data - Deep Dive
- ▶ *Reducing data dimension by Random Projection* recipe in Chapter 4, Analyzing Data - Deep Dive
- ▶ *Building Decision Trees to solve Multi Class Problems* recipe in Chapter 6, Machine Learning I
- ▶ *Understanding Ensemble, Gradient Boosting* recipe in Chapter 8, Model Selection and Evaluation
- ▶ *Growing from trees to Forest, Random Forest* recipe in Chapter 9, Machine Learning III
- ▶ *Growing Extremely Randomized Trees* recipe in Chapter 9, Machine Learning III

10

Large-Scale Machine Learning – Online Learning

In this chapter, we will see the following recipes:

- ▶ Using perceptron as an online linear algorithm
- ▶ Using stochastic gradient descent for regression
- ▶ Using stochastic gradient descent for classification

Introduction

In this chapter, we will concentrate on large-scale machine learning and the algorithms suited to tackle such large-scale problems. Till now, when we trained all our models, we assumed that our training set can fit into our computer's memory. In this chapter, we will see how to go about building models when this assumption is no longer satisfied. Our training records are of a huge size and so we cannot fit them completely into our memory. We may have to load them piecewise and still produce a model with a good accuracy. The argument of a training set not fitting into our computer memory can be extrapolated to streaming data. With streaming data, we don't see all the data at once. We should be able to make decisions based on whatever data we are exposed to and also have a mechanism for continuously improving our model as new data arrives.

We will introduce the framework of the stochastic gradient descent-based algorithms. This is a versatile framework to handle very large-scale datasets that will not fit completely into our memory. Several types of linear algorithms, including logistic regression, linear regression, and linear SVM, can be accommodated using this framework. The kernel trick, which we introduced in our previous chapter, can be included in this framework in order to deal with datasets with nonlinear relationships.

We will begin our list of recipes with the perceptron algorithm, the oldest machine learning algorithm. Perceptron is easy to understand and implement. However, Perceptron is limited to solving only linear problems. A kernel-based perceptron can be used to solve nonlinear datasets.

In our second recipe, we will formally introduce the framework of gradient descent-based methods and how it can be used to perform regression-based tasks. We will look at different loss functions to see how different types of linear models can be built using these functions. We will also see how perceptron belongs to the family of stochastic gradient descent.

In our final recipe, we will see how classification algorithms can be built using the stochastic gradient descent framework.

Even though we don't have a direct example of streaming data, with our existing datasets, we will see how the streaming data use cases can be addressed. Online learning algorithms are not limited to streaming data, they can be applied to batch data also, except that they process only one instance at a time.

Using perceptron as an online learning algorithm

As mentioned earlier, perceptron is one of the oldest machine learning algorithms. It was first mentioned in a 1943 paper:

A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY. WARREN S. MCCULLOCH AND WALTER PITTS University of Illinois, College of Medicine, Department of Psychiatry at the Illinois Neuropsychiatric Institute, University of Chicago, Chicago, U.S.A.

Let's revisit our definition of a classification problem. Each record or instance can be written as a set (X,y) , where X is a set of attributes and y is a corresponding class label.

Learning a target function, F , that maps each record's attribute set to one of the predefined class label, y , is the job of a classification algorithm.

The difference in our case is that we have a large-scale learning problem. All our data will not fit into our main memory. So, we need to keep our data on a disk and use only a portion of it at a time in order to build our perceptron model.

Let's proceed to outline the perceptron algorithm:

1. Initialize the weights of the model to a small random number.
2. Center the input data, x , with its mean.
3. At each time step t (also called epoch):
 - Shuffle the dataset
 - Pick a single instance of the record and make a prediction
 - Observe the deviation of the prediction from the true label output
 - Update the weights if the prediction is different from the true label

Let's consider the following scenario. We have the complete dataset on our disk. In a single epoch, that is, in step 3, all the steps mentioned are performed on all the data on our disk. In an online learning scenario, a bunch of instances based on a windowing function will be available to us at any point in time. We can update the weights as many times as the number of instances in our window in a single epoch.

Let's see how to go about updating our weights.

Let's say our input X is as follows:

$$X_i = \{x_1, x_2, x_3, \dots, x_m\}, \text{ where } i = 1 \text{ to } n$$

Our Y is as follows:

$$Y = \{+1, -1\}$$

We will define our weights as the following equation:

$$W = \{w_1, w_2, w_3, \dots, w_m\}$$

Our prediction after we see each record is defined as follows:

$$\hat{y}_i = \text{sign}(w_i * x_i)$$

The sign function returns +1 if the product of the weight and attributes is positive, or -1 if the product is negative.

Perceptron proceeds to compare the predicted y with the actual y . If the predicted y is correct, it moves on to the next record. If the prediction is incorrect, there are two scenarios. If the predicted y is $+1$ and the actual y is -1 , it decrements the weight with an x value, and vice versa. If the actual y is $+1$ and the predicted y is -1 , it increments the weights. Let's see this as an equation for more clarity:

$$w_{t+1} = w_t + y_i x_i$$

Typically, a learning rate alpha is provided so that the weights are updated in a controlled manner. With the presence of noise in the data, a full increment of decrements will lead to the weights not converging:

$$w_{t+1} = w_t + \alpha(y_i x_i)$$

Alpha is a very small value ranging, between 0.1 and 0.4.

Let's jump into our recipe now.

Getting ready

Let's generate data using `make_classification` in batches with a generator function to simulate large-scale data and data streaming, and proceed to write the perceptron algorithm.

How to do it...

Let's load the necessary libraries. We will then write a function, `get_data`, which is a generator:

```
from sklearn.datasets import make_classification
from sklearn.metrics import classification_report
from sklearn.preprocessing import scale
import numpy as np

def get_data(batch_size):
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    b_size = 0
    no_features = 30
    redundant_features = int(0.1*no_features)
```

```

informative_features = int(0.8*no_features)
repeated_features = int(0.1*no_features)

while b_size < batch_size:
    x,y = make_classification(n_samples=1000,n_features=no_
features,flip_y=0.03,\n_informative = informative_features, n_redundant =
redundant_features \
, n_repeated = repeated_features, random_state=51)
    y_indx = y < 1
    y[y_indx] = -1
    x = scale(x,with_mean=True,with_std=True)

    yield x,y
    b_size+=1

```

We will proceed to write two functions, one to build our perceptron model and the other one to test the worthiness of our model:

```

def build_model(x,y,weights,epochs,alpha=0.5):
    """
    Simple Perceptron
    """

    for i in range(epochs):

        # Shuffle the dataset
        shuff_index = np.random.shuffle(range(len(y)))
        x_train = x[shuff_index,:].reshape(x.shape)
        y_train = np.ravel(y[shuff_index,:])

        # Build weights one instance at a time
        for index in range(len(y)):
            prediction = np.sign( np.sum(x_train[index,:] * weights) )
            if prediction != y_train[index]:
                weights = weights + alpha * (y_train[index] * x_
train[index,:])

    return weights

def model_worth(x,y,weights):
    prediction = np.sign(np.sum(x * weights, axis=1))
    print classification_report(y,prediction)

```

Finally, we will write our main function to invoke all the preceding functions, to demonstrate the perceptron algorithm:

```
if __name__ == "__main__":
    data = get_data(10)
    x,y = data.next()
    weights = np.zeros(x.shape[1])
    for i in range(10):
        epochs = 100
        weights = build_model(x,y,weights,epochs)
        print
        print "Model worth after receiving dataset batch %d" %(i+1)
        model_worth(x,y,weights)
        print
        if i < 9:
            x,y = data.next()
```

How it works...

Let's start with our main function. We will ask our generator to send us 10 sets of data:

```
data = get_data(10)
```

Here, we want to simulate both large-scale data and data streaming. While building our model, we don't have access to all the data, just part of it:

```
x,y = data.next()
```

We will use the `next()` function in the generator in order to get the next set of data. In the `get_data` function, we will use the `make_classification` function from scikit-learn:

```
x,y = make_classification(n_samples=1000,n_features=no_
features,flip_y=0.03,\
n_informative = informative_features, n_redundant =
redundant_features \
,n_repeated = repeated_features, random_state=51)
```

Let's look at the parameters passed to the `make_classification` method. The first parameter is the number of instances required, in this case, we need 1,000 instances. The second parameter is about how many attributes per instance are required. We will assume that we need 30. The third parameter, `flip_y`, randomly interchanges 3 percent of the instances. This is done to introduce some noise in our data. The next parameter is about these 30 features and how many of them should be informative enough to be used in our classification. We specified that 60 percent of our features, that is, 18 out of 30, should be informative. The next parameter is about the redundant features. These are generated as a linear combination of the informative features in order to introduce correlation among the features. Finally, repeated features are duplicate features that are drawn randomly from both the informative features and the redundant features.

When we call `next()`, we will get 1,000 instances of this data. This function returns a y label as `{0, 1}`; we want `{-1, +1}` and hence we will change all the zeros in y to -1:

```
y_indx = y < 1  
y[y_indx] = -1
```

Finally, we will center our data using the `scale` function from scikit-learn.

Let's proceed to build our model with the first batch of data. We will initialize our weights matrix with zeros:

```
weights = np.zeros(x.shape[1])
```

As we need 10 batches of data to simulate large-scale learning and data streaming, we will do the model building 10 times in the for loop:

```
for i in range(10):  
    epochs = 100  
    weights = build_model(x,y,weights,epochs)
```

Our perceptron algorithm is built in `build_model`. A predictor x, response variable y, the weights matrix, and number of time steps or epochs are passed as parameters. In our case, we have set the number of epochs to 100. This function has one additional parameter, alpha value:

```
def build_model(x,y,weights,epochs,alpha=0.5)
```

By default, we have set our alpha value to 0.5.

Let's see in our `build_model`. We will start with shuffling the data:

```
# Shuffle the dataset  
shuff_index = np.random.shuffle(range(len(y)))  
x_train = x[shuff_index,:].reshape(x.shape)  
y_train = np.ravel(y[shuff_index,:])
```

We will go through each record in our dataset and start updating our weights:

```
# Build weights one instance at a time  
for index in range(len(y)):  
    prediction = np.sign( np.sum(x_train[index,:] * weights) )  
    if prediction != y_train[index]:  
        weights = weights + alpha * (y_train[index] * x_  
train[index,:])
```

In the for loop, you can see that we do the prediction:

```
prediction = np.sign( np.sum(x_train[index,:] * weights) )
```

We will multiply our training data with weights, and add them together. Finally, we will use the `np.sign` function to get our prediction. Now, based on the prediction, we will update our weights:

```
weights = weights + alpha * (y_train[index] * x_train[index,:])
```

That is all. We will return the weights to the calling function.

In our main function, we will invoke the `model_worth` function to print the goodness of the model. Here, we will use the `classification_report` convenience function to print the accuracy score of the model:

```
print
print "Model worth after receiving dataset batch %d" %(i+1)
model_worth(x,y,weights)
```

We will then proceed to update our model for the next batch of incoming data. Note that we have not altered the `weights` parameter. It gets updated with every batch of new data coming in.

Let's see what `model_worth` has printed:

Model worth after receiving dataset batch 1				
	precision	recall	f1-score	support
-1	0.76	0.77	0.77	499
1	0.77	0.76	0.76	501
avg / total	0.77	0.77	0.76	1000

Model worth after receiving dataset batch 2				
	precision	recall	f1-score	support
-1	0.74	0.77	0.76	499
1	0.76	0.73	0.74	501
avg / total	0.75	0.75	0.75	1000

Model worth after receiving dataset batch 3				
	precision	recall	f1-score	support
-1	0.75	0.76	0.76	499
1	0.76	0.75	0.76	501
avg / total	0.76	0.76	0.76	1000

Model worth after receiving dataset batch 4				
	precision	recall	f1-score	support
-1	0.76	0.75	0.76	499
1	0.76	0.76	0.76	501
avg / total	0.76	0.76	0.76	1000

There's more...

Scikit-learn provides us with an implementation of perceptron. Refer to the following URL for more details:

http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html.

Another improvement that can be made in the perceptron algorithm is to use more features.

Remember the prediction equation, we can rewrite it as follows:

$$\hat{y} = \text{sign}(w_i * \phi(x_i))$$

We replaced the x values with a function. Here, we can send a feature generator. For example, a polynomial feature generator can be added to our `get_data` function, as follows:

```
def get_data(batch_size):
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    b_size = 0
    no_features = 30
    redundant_features = int(0.1*no_features)
    informative_features = int(0.8*no_features)
    repeated_features = int(0.1*no_features)
    poly = PolynomialFeatures(degree=2)

    while b_size < batch_size:
        x,y = make_classification(n_samples=1000,n_features=no_
features,flip_y=0.03,\n            n_informative = informative_features, n_redundant =
redundant_features \
            ,n_repeated = repeated_features, random_state=51)
        y_indx = y < 1
        y[y_indx] = -1
        x = poly.fit_transform(x)
        yield x,y
        b_size+=1
```

Finally, kernel-based perceptron algorithms are available to handle nonlinear datasets. Refer to the Wikipedia article for more information about kernel-based perceptron:

https://en.wikipedia.org/wiki/Kernel_perceptron.

See also

- ▶ *Learning and using Kernels* recipe in Chapter 5, Data Mining - Finding a needle in a haystack

Using stochastic gradient descent for regression

In a typical regression setup, we have a set of predictors (instances), as follows:

$$X = \{x_1, x_2, \dots, x_n\}$$

Each instance has m attributes, as follows:

$$x_i = \{x_{i1}, x_{i2}, \dots, x_{im}\} \text{ where } i = 1 \text{ to } n$$

The response variable, Y, is a vector of real-valued entries. The job of regression is to find a function such that when x is provided as an input to this function, it should return y:

$$F(X) = Y$$

The preceding function is parameterized by a weight vector, that is, a combination of the weight vector and input vector is used to predict y, so rewriting the function with the weight vector will get the following:

$$F(X, W) = Y$$

So, the question now is how do we know that we have the right weight vectors? We will use a loss function, L, to get the right weight vectors. The loss function measures the cost of making a wrong prediction. It empirically measures the cost of predicting y when the actual value is y. The regression problem now becomes the problem of finding the right weight vector that will minimize the loss function. For our whole dataset of n elements, the overall loss function is as follows:

$$\frac{1}{n} \sum_{i=1}^n L(f(x_i, w), y_i)$$

Our weight vectors should be those that minimize the preceding value.

Gradient descent is an optimization technique used to minimize the preceding equation. For this equation, we will find the gradient, that is, the first-order derivative with respect to W.

Unlike other optimization techniques such as the batch gradient descent, stochastic gradient descent operates on one instance at a time. The steps involved in stochastic gradient descent are as follows:

1. For each epoch, shuffle the dataset.
2. Pick an instance and its response variable, y.
3. Calculate the loss function and its derivative, w.r.t weights.
4. Update the weights.

Let's say:

$$\nabla_w$$

This signifies the derivative, w.r.t w. The weights are updated as follows:

$$w_{i+1} = w_i - \nabla_w L(f(x_i, w), y_i)$$

As you can see, the weights are moved in the opposite direction to the gradient, thus forcing a descent that will eventually give the weight vector values, which can reduce the objective cost function.

A squared loss is a typical loss function used with regression. The squared loss of an instance is defined in the following way:

$$(\hat{y} - y)^2$$

The derivative of the preceding equation is substituted into the weight update equation. With this background knowledge, let's proceed to our recipe for stochastic gradient descent regression.

As explained in perceptron, a learning rate, eta, is added to the weight update equation in order to avoid the effect of noise:

$$w_{i+1} = w_i - \eta (\nabla_w L(f(x_i, w), y_i))$$

Getting ready

We will be leveraging the scikit-learn's implementation of SGD regression. As in some of the previous recipes, we will use the `make_regression` function from scikit-learn to generate data for our recipe in order to demonstrate stochastic gradient descent regression.

How to do it...

Let's start with a very simple example demonstrating how to build a stochastic gradient descent regressor.

We will first load the required libraries. We will then write a function to generate predictors and response variables to demonstrate regression:

```
from sklearn.datasets import make_regression
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.cross_validation import train_test_split

def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 30

    x,y = make_regression(n_samples=1000,n_features=no_features,\n                          random_state=51)
    return x,y
```

We will proceed to write the functions that will help us build, validate, and inspect our model:

```
def build_model(x,y):
    estimator = SGDRegressor(n_iter = 10, shuffle=True, loss = "squared_
loss", \
                            learning_rate='constant', eta0=0.01, fit_intercept=True, \
                            penalty='none')
    estimator.fit(x,y)

    return estimator

def model_worth(model,x,y):
    predicted_y = model.predict(x)
```

```

        print "\nMean absolute error = %0.2f"%mean_absolute_
error(y,predicted_y)
        print "Mean squared error = %0.2f"%mean_squared_
error(y,predicted_y)

def inspect_model(model):
    print "\nModel Intercept {0}".format(model.intercept_)
    print
    for i,coef in enumerate(model.coef_):
        print "Coefficient {0} = {1:.3f}".format(i+1,coef)

```

Finally, we will write our main function to invoke all the preceding functions:

```

if __name__ == "__main__":
    x,y = get_data()

    # Divide the data into Train, dev and test
    x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
    x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)

    model = build_model(x_train,y_train)

    inspect_model(model)

    print "Model worth on train data"
    model_worth(model,x_train,y_train)
    print "Model worth on dev data"
    model_worth(model,x_dev,y_dev)

    # Building model with l2 regularization
    model = build_model_regularized(x_train,y_train)
    inspect_model(model)

```

How it works...

Let's start with our main function. We will invoke the `get_data` function to generate our predictor `x` and response variable `y`:

```
x,y = get_data()
```

In the `get_data` function, we will leverage the convenient `make_regression` function from scikit-learn to generate a dataset for the regression problems:

```

no_features = 30
x,y = make_regression(n_samples=1000,n_features=no_features,\n
random_state=51)

```

As you can see, we will generate a dataset with 1,000 instances specified by an `n_samples` parameter, and 30 features defined by an `n_features` parameter.

Let's split the data into training and testing sets using `train_test_split`. We will reserve 30 percent of our data to test:

```
# Divide the data into Train, dev and test
x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
```

Once again, we will leverage `train_test_split` to split our test data into dev and test sets:

```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)
```

With the data divided to build, evaluate, and test the model, we will proceed to build our models.

We will invoke the `build_model` function with our training dataset:

```
model = build_model(x_train,y_train)
```

In `build_model`, we will leverage scikit-learn's SGD regressor class to build our stochastic gradient descent method:

```
estimator = SGDRegressor(n_iter = 10, shuffle=True,loss = "squared_
loss", \
learning_rate='constant',eta0=0.01,fit_intercept=True, \
penalty='none')
estimator.fit(x,y)
```

The SGD regressor is a vast method and can be used to fit a number of linear models with a lot of parameters. We will first explain the basic method of stochastic gradient descent and then proceed to explain the other details.

Let's look at the parameters that we used. The first parameter is the number of times that we want to go through our dataset in order to update the weights. Here, we will say that we want 10 iterations. As in perceptron, after going through all the records once, we need to shuffle our input records when we start the next iteration. A parameter `shuffle` is used for the same. The default value of `shuffle` is true, we have included it here for explanation purposes. Our loss function is the squared loss and we want to do a linear regression; hence, we will specify this using the `loss` parameter.

Our learning rate, `eta`, is a constant that we will specify with the `learning_rate` parameter. We will provide a value for our learning rate using the `eta0` parameter. We will then say that we need to fit the intercept as we have not centered our data by its mean. Finally, the `penalty` parameter controls the type of shrinkage required. In our case, we don't need any shrinkage using the `none` string.

We will proceed to build our model by invoking the fit function with our predictor and response variable. Finally we will return the model that we built to our calling function.

Let's now inspect our model and see the value of the intercept and coefficients:

```
inspect_model(model)
```

In the inspect model, we will print the values of the model intercepts and coefficients:

```
Model Intercept [ 0.00116188]

Coefficient 1 = 41.822
Coefficient 2 = -0.002
Coefficient 3 = 73.805
Coefficient 4 = -0.001
Coefficient 5 = 0.002
Coefficient 6 = 58.607
Coefficient 7 = 0.002
Coefficient 8 = 0.001
Coefficient 9 = -0.004
Coefficient 10 = 0.000
Coefficient 11 = 9.144
Coefficient 12 = 0.002
Coefficient 13 = 0.000
Coefficient 14 = 63.512
Coefficient 15 = -0.001
Coefficient 16 = 0.004
Coefficient 17 = -0.000
Coefficient 18 = -0.001
Coefficient 19 = -0.000
Coefficient 20 = -0.002
Coefficient 21 = 98.772
Coefficient 22 = 3.833
Coefficient 23 = -0.003
Coefficient 24 = 99.226
Coefficient 25 = -0.001
Coefficient 26 = -0.001
Coefficient 27 = 0.002
Coefficient 28 = 56.238
Coefficient 29 = -0.002
Coefficient 30 = 84.769
```

Let's now look at how our model has performed in our training data:

```
print "Model worth on train data"
model_worth(model,x_train,y_train)
```

We will invoke the model_worth function to look at our model's performance. The model_worth function prints the mean absolute error and mean squared error values.

The mean squared error is defined as follows:

$$\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

The mean absolute error is defined in the following way:

$$\frac{1}{n} \sum_{i=1}^n |\hat{y}_i - y_i|$$

The mean squared error is sensitive to outliers. Hence, the mean absolute error is a more robust measure. Let's look at the model's performance using the training data:

```
Model worth on train data
Mean absolute error = 0.02
Mean squared error = 0.00
```

Let's now look at the model's performance using our dev data:

```
Model worth on dev data
Mean absolute error = 0.00
Mean squared error = 0.00
```

There's more...

We can include regularization in the stochastic gradient descent framework. Recall the following cost function of ridge regression from the previous chapter:

$$\sum_{i=1}^n \left(y_i - w_0 - \sum_{j=1}^m x_{ij} w_{ij} \right)^2 + \alpha \sum_{j=1}^m w_j^2$$

We included an expanded version of the square loss function here and added the regularization term—the sum of the square of the weights. We can include it in our gradient descent procedure. Let's say that we denote our regularization term as $R(W)$. Our weight update is now as follows:

$$w_{i+1} = w_i - \eta \left(\nabla_w L(f(x_i, w), y_i) + \alpha (\nabla_w R(w)) \right)$$

As you can see, now we have the derivative of the loss function with respect to the weight vector, w , and the derivative of the regularization term with respect to the weights is added to our weight update rule.

Let's write a new function to build our model to include regularization:

```
def build_model_regularized(x, y):
    estimator = SGDRegressor(n_iter = 10, shuffle=True, loss = "squared_
loss", \
                            learning_rate='constant', eta0=0.01, fit_intercept=True, \
                            penalty='l2', alpha=0.01)
    estimator.fit(x, y)

    return estimator
```

We can invoke this function from our main function as follows:

```
model = build_model_regularized(x_train, y_train)
inspect_model(model)
```

Let's see the new parameters that we passed compared with our previous build model method:

```
estimator = SGDRegressor(n_iter = 10, shuffle=True, loss = "squared_
loss", \
                        learning_rate='constant', eta0=0.01, fit_intercept=True, \
                        penalty='l2', alpha=0.01)
```

Earlier, we mentioned our penalty as none. Now, you can see that we mentioned that we need to add an L2 penalty to our model. Again, we will give an alpha value of 0.01 using the alpha parameter. Let's look at our coefficients:

```
Model Intercept [ 3.49353155e-15]

Coefficient 1 = 41.826
Coefficient 2 = 0.000
Coefficient 3 = 73.811
Coefficient 4 = -0.000
Coefficient 5 = -0.000
Coefficient 6 = 58.613
Coefficient 7 = 0.000
Coefficient 8 = 0.000
Coefficient 9 = 0.000
Coefficient 10 = -0.000
Coefficient 11 = 9.147
Coefficient 12 = -0.000
Coefficient 13 = 0.000
Coefficient 14 = 63.520
Coefficient 15 = 0.000
Coefficient 16 = 0.000
Coefficient 17 = 0.000
Coefficient 18 = 0.000
Coefficient 19 = 0.000
Coefficient 20 = -0.000
Coefficient 21 = 98.782
Coefficient 22 = 3.832
Coefficient 23 = -0.000
Coefficient 24 = 99.239
Coefficient 25 = 0.000
Coefficient 26 = 0.000
Coefficient 27 = -0.000
Coefficient 28 = 56.245
Coefficient 29 = 0.000
Coefficient 30 = 84.775
```

You can see the effect of the L2 regularization: a lot of the coefficients have attained a zero value. Similarly, the L1 regularization and elastic net, which combines both the L1 and L2 regularization, can be included using the penalty parameter.

Remember in our introduction, we mentioned that stochastic gradient descent is more of a framework than a single method. Other linear models can be generated using this framework by changing the loss function.

SVM regression models can be built using the epsilon-insensitive loss function. This loss function is defined as follows:

$$L(f(x_i, w), y_i) = \begin{cases} 0 & \text{if } |y_i - f(x_i, w)| < \epsilon \\ |y_i - f(x_i, w)| - \epsilon & \text{otherwise} \end{cases}$$

Refer to the following URL for the various parameters that can be passed to the SGD regressor in scikit-learn:

http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html.

See also

- ▶ *Predicting real valued numbers using regression recipe in Chapter 7, Machine Learning II*
- ▶ *Shrinkage using Ridge Regression recipe in Chapter 7, Machine Learning II*

Using stochastic gradient descent for classification

A classification problem setup is very similar to a regression setup except for the response variable. In a classification setup, the response is a categorical variable. Due to its nature, we have a different loss function to measure the cost of the wrong predictions. Let's assume a binary classifier for our discussion and recipe, and our target variable, Y, can take the values {0,1}.

We will use the derivative of this loss function in our weight update rule to arrive at our weight vectors.

The SGD classifier class from scikit-learn provides us with a variety of loss functions. However, in this recipe, we will see log loss, which will give us logistic regression.

Logistic regression fits a linear model to a data of the following form:

$$W^T X$$

We have given a generalized notation. The intercept is assumed to be the first dimension of our weight vector. For a binary classification problem, a logit function is applied to get a prediction. as follows:

$$F(w, x_i) = \frac{1}{1 + e^{-w^T x_i}}$$

The preceding function is also called the sigmoid function. For very large positive values of x_i , this function will return a value close to one, and vice versa for large negative values close to zero. With this, we can define our log loss function as follows:

$$L(w, x_i) = -y_i \log(F(w, x_i)) - (1 - y_i) \log(1 - F(w, x_i))$$

With the preceding loss function fitted into the weight update rule of the gradient descent, we can arrive at the appropriate weight vectors.

For the log loss function defined in scikit-learn, refer to the following URL:

http://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html.

With this knowledge, let's jump into our recipe for stochastic gradient descent-based classification.

Getting ready

We will leverage scikit-learn's implementation of the stochastic gradient descent classifier. As we did in some of the previous recipes, we will use the `make_classification` function from scikit-learn to generate data for our recipe in order to demonstrate the stochastic gradient descent classification.

How to do it...

Let's start with a very simple example demonstrating how to build a stochastic gradient descent regressor.

We will first load the required libraries. We will then write a function to generate the predictors and response variables:

```
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import SGDClassifier

import numpy as np

def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 30
    redundant_features = int(0.1*no_features)
    informative_features = int(0.6*no_features)
    repeated_features = int(0.1*no_features)
    x,y = make_classification(n_samples=1000,n_features=no_
    features,flip_y=0.03,\n        n_informative = informative_features, n_redundant =
    redundant_features \
        ,n_repeated = repeated_features,random_state=7)
    return x,y
```

We will proceed to write functions that will help us build and validate our model:

```
def build_model(x,y,x_dev,y_dev):
    estimator = SGDClassifier(n_iter=50,shuffle=True,loss="log", \
        learning_rate = "constant",eta0=0.0001,fit_\
    intercept=True, penalty="none")
    estimator.fit(x,y)
    train_predcited = estimator.predict(x)
    train_score = accuracy_score(y,train_predcited)
    dev_predicted = estimator.predict(x_dev)
    dev_score = accuracy_score(y_dev,dev_predicted)

    print
    print "Training Accuracy = %0.2f Dev Accuracy = %0.2f"%(train_\
    score,dev_score)
```

Finally, we will write our main function to invoke all the preceding functions:

```
if __name__ == "__main__":
    x,y = get_data()
```

```
# Divide the data into Train, dev and test
x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_
size = 0.3,random_state=9)
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_
all,test_size=0.3,random_state=9)

build_model(x_train,y_train,x_dev,y_dev)
```

How it works...

Let's start with our main function. We will invoke `get_data` to get our `x` predictor attributes and `y` response attributes. In `get_data`, we will leverage the `make_classification` dataset in order to generate our training data for the random forest method:

```
def get_data():
    """
    Make a sample classification dataset
    Returns : Independent variable y, dependent variable x
    """
    no_features = 30
    redundant_features = int(0.1*no_features)
    informative_features = int(0.6*no_features)
    repeated_features = int(0.1*no_features)
    x,y = make_classification(n_samples=500,n_features=no_features,flip_
y=0.03,\n        n_informative = informative_features, n_redundant =
redundant_features \
        ,n_repeated = repeated_features,random_state=7)
    return x,y
```

Let's look at the parameters passed to the `make_classification` method. The first parameter is the number of instances required. In this case, we need 500 instances. The second parameter is about how many attributes per instance are required. We say that we need 30. The third parameter, `flip_y`, randomly interchanges 3 percent of the instances. This is done to introduce noise in our data. The next parameter is about how many out of those 30 features should be informative enough to be used in our classification. We specified that 60 percent of our features, that is, 18 out of 30, should be informative. The next parameter is about redundant features. These are generated as a linear combination of the informative features in order to introduce correlation among the features. Finally, the repeated features are duplicate features that are drawn randomly from both the informative and redundant features.

Let's split the data into training and testing sets using `train_test_split`. We will reserve 30 percent of our data to test:

```
# Divide the data into Train, dev and test
x_train,x_test_all,y_train,y_test_all = train_test_split(x,y,test_size = 0.3,random_state=9)
```

Once again, we will leverage `train_test_split` to split our test data into dev and test sets:

```
x_dev,x_test,y_dev,y_test = train_test_split(x_test_all,y_test_all,test_size=0.3,random_state=9)
```

With the data divided to build, evaluate, and test the model, we will proceed to build our models:

```
build_model(x_train,y_train,x_dev,y_dev)
```

In `build_model`, we will leverage scikit-learn's `SGDClassifier` class to build our stochastic gradient descent method:

```
estimator = SGDClassifier(n_iter=50,shuffle=True,loss="log", \
                           learning_rate = "constant",eta0=0.0001,fit_intercept=True, penalty="none")
```

Let's look at the parameters that we used. The first parameter is the number of times we want to go through our dataset to update the weights. Here, we say that we want 50 iterations. As in perceptron, after going through all the records once, we need to shuffle our input records when we start the next iteration. The shuffle parameter is used for the same. The default value of shuffle is true, we have included it here for explanation purposes. Our loss function is log loss: we want to do a logistic regression and we will specify this using the loss parameter. Our learning rate, eta, is a constant that we will specify with the `learning_rate` parameter. We will provide the value for our learning rate using the `eta0` parameter. We will then proceed to say that we need to fit the intercept, as we have not centered our data by its mean. Finally, the penalty parameter controls the type of shrinkage required. In our case, we will say that we don't need any shrinkage using the none string.

We will proceed to build our model by invoking the `fit` function with our predictor and response variable, and evaluate our model with our training and dev dataset:

```
estimator.fit(x,y)
train_predcited = estimator.predict(x)
train_score = accuracy_score(y,train_predcited)
dev_predicted = estimator.predict(x_dev)
dev_score = accuracy_score(y_dev,dev_predicted)

print
print "Training Accuracy = %0.2f Dev Accuracy = %0.2f"%(train_score,dev_score)
```

Let's look at our accuracy scores:

Training Accuracy = 0.83 Dev Accuracy = 0.82

There's more...

Regularization, L1, L2, or elastic net can be applied for SGD classification. The procedure is the same as that of regression, and hence, we will not repeat it here. Refer to the previous recipe for this.

The learning rate, eta, was constant in our example. This need not be the case. With every iteration, the eta value can be reduced. The learning rate parameter, `learning_rate`, can be set to an optimal string or `invscaling`. Refer to the following scikit documentation:

<http://scikit-learn.org/stable/modules/sgd.html>.

The parameter is specified as follows:

```
estimator = SGDClassifier(n_iter=50, shuffle=True, loss="log", \
    learning_rate = "invscaling", eta0=0.001, fit_intercept=True, \
    penalty="none")
```

We used the `fit` method to build our model. As mentioned previously, in large-scale machine learning, we know that all the data will not be available to us at once. When we receive the data in batches, we need to use the `partial_fit` method, instead of `fit`. Using the `fit` method will reinitialize the weights and we will lose all the training information from the previous batch of data. Refer to the following link for more information on `partial_fit`:

http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html#sklearn.linear_model.SGDClassifier.partial_fit.

See also

- ▶ *Shrinkage using Ridge Regression* recipe in *Chapter 7, Machine Learning II*
- ▶ *Using stochastic gradient descent for regression* recipe in *Chapter 9, Machine Learning III*

Index

A

accuracy 234
AdaBoost 326
anonymous functions
 creating, with lambda 34
orange function
 about 60
 URL 60
arrays
 processing, from tabular data 39-42
axioms
 about 186
 URL 186

B

Bagging
 about 316, 317
 leveraging 317-325
BaseEstimator
 URL 82
Boosting
 about 316, 325
 two-class classification problem 325-340
Bootstrapping 317
box-and-whisker plot
 about 114
 using 114-116
built-in morphy-function
 reference link 139

C

classification
 data, preparing 228-233
 stochastic gradient descent, using 405-409

columns
 preprocessing 43, 44
comprehension 19
confusion matrix 234
cost function 269
counter
 about 5
 reference link 5
CountVectorizer class
 about 146
 reference link 146
cross-validation iterators
 reference link 313
 used, with L1 and L2 shrinkage 301-313
csv library
 URL 42
curse of dimensionality 152

D

data
 clustering, k-means method used 196-202
 dimension, reducing with random
 projection 171-174
 grouping 95-99
 imputing 117-119
 preparing, for classification model 228-233
 scaling 122-124
 standardizing 125, 126
data model
 URL 3
data preprocessing 86
dataset
 URL 77

decision trees
advantages 256
building, for multiclass problems 255-265
disadvantages 256
reference link 255

decorators

used, for altering function behavior 31-33

deque

about 19
URL 19

derivational patterns

reference link 138

dictionaries

about 5
dictionary objects, using 2-5
dictionary of dictionaries, using 6, 7
URL 4

dimensionality reduction 152

distance measures

calculating 187-189
URL 192
working with 186-191

documents

classifying, Naïve Bayes used 242-254

dot plots

using 95-99

E

ensemble methods

Bagging 317-325
Boosting 325-340
Gradient Boosting 341

error rate 234

Exploratory Data Analysis (EDA) 86

ExtraTreesClassifier class

reference link 374

Extremely Randomized Trees

about 369
implementing 369-375

F

feature matrices

decomposing, Non-negative Matrix Factorization (NMF) used 175-183

feature test condition 257

filters

using 36

function

behavior altering, with decorators 31-33
embedding, in another function 28
passing, as parameter 29
passing, as variable 27
returning 30, 31

functools

about 31
URL 31

G

generator

generating 24, 25

Gradient Boosting

about 341
demonstrating 343-354
reference link 354
simple regression problem 341, 342

Graphviz package

URL 265

H

heat maps

URL 104
using 104-108

I

information gain

about 259
reference link 259

instance-based learning 235

inverse document frequencies

calculating 147-150

iterables

using 26, 27

iterator

generating 24, 25
reference link 24
using 22, 23

itertools

about 51
URL 53
using 51-53

Itertools.dropwhile

reference link 22

izip

using 37-39

K**kernel-based perceptron**

URL 395

kernel methods

learning 192-196

linear kernel 196

polynomial kernel 196

using 192-196

kernel PCA

using 160-166

key

used, for sorting 46-51

k-fold cross-validation 301**k-means method**

cluster evaluation, measures 197

used, for data clustering 196-202

K-Nearest Neighbor (KNN) 235**L****L1 shrinkage (LASSO)**

used, with regression 293-300

L2 shrinkage (ridge)

used, with regression 283-292

lambda

used, for creating anonymous functions 34

Last In, First Out (LIFO) 18**Latent Semantic Analysis (LSA)**

about 170

reference link 170

lazy learner 235**Least absolute shrinkage and selection**

operator (LASSO) 293

least squares

reference link 270

lemmatization 138**linear kernel**

about 196

URL 196

list

list comprehension, creating 19-21

sorting 45, 46

writing 15-18

loadtxt method

about 42

reference link 42

Local Outlier Factor (LOF)

used, for discovering outliers 216-225

M**machine learning**

with scikit-learn 75-84

map function

using 35

matplotlib

about 55

plotting with 65-74

URL 74

matrix decomposition 152**multiclass problems**

solving, with decision trees 255-265

multivariate data

scatter plots, using 100-103

N**Naïve Bayes**

used, for classifying documents 242-254

namedtuple

URL 12

nearest neighbors

obtaining 234-241

Non-negative Matrix Factorization (NMF)

reference link 175

used, for decomposing feature

matrices 175-183

NumPy libraries

object, properties 59

using 55-64

O**online learning algorithm**

perceptron, using 388-395

OrderedDict container
about 4
URL 4

outliers
discovering, local outlier factor method
used 216-225
finding, in univariate data 208-216

out-of-bag estimation (OOB)
about 368
reference link 368

P

pairwise_distance method
about 192
URL 192

partial_fit method
about 410
URL 410

percentiles, NumPy
reference link 92

perceptron
reference link 395
used, as online learning algorithm 388-395

polynomial kernel
about 196
URL 196

polysemy 170

Principal Component Analysis (PCA) 153

principal components
extracting 153-159

priority queues
URL 220

progressive sampling 122

pypot
about 87
reference link 87
URL 74

Python NLTK library
URL 7

R

Radial Basis Function (RBF) kernel
about 164
reference link 164

Random Forest
about 358
implementing 359-368

RandomForestClassifier class
about 364
reference link 364

randomization 317

random projection
data dimension, reducing with 171-174
reference link 174

random sampling
performing 120, 121

real-valued numbers
predicting, regression used 268-282

recursive feature selection 279

regression
about 268
stochastic gradient descent, using 396-405
used, for predicting real-valued
numbers 268-282
with L1 shrinkage (LASSO) 293-300
with L2 shrinkage (ridge) 283-292

ridge regression 284

Rotational Forest
about 376
building 376-384

rote classifier algorithm 235

S

scatter plots
used, for multivariate data 100-103

scikit-learn
machine learning with 75-84
URL 84

SciPy
URL, for documentation 45

sent_tokenize function
URL 128
using 128

set builder notation
reference link 22

sets
using 12-14

shrinkage 283

Singular Value Decomposition (SVD)
used, for extracting features 166-170

snowball stemmers
reference link 137

sparse matrix representation
reference link 143

standardization 124

star convex-shaped data points
URL 202

stemming, words
performing 135-137

stochastic gradient descent
used, for classification 405-409
used, for regression 396-405

stop words
removing 131-134

stratified sampling 121

summary statistics
performing 109-113
plotting 109-113

T

tabular data
arrays, processing 39-42

term frequencies
calculating 147-150

Term Frequency Inverse Document Frequency (TFIDF) 169

text
representing, as bag of words 140-146

text mining
reference link 133

TFIDF transformer
calculating 150

tokenization
performing 127-130

tuples
about 7
creating 8-12
manipulating 8-12

U

univariate data
analyzing, graphically 87-93
outliers, finding 208-216

V

vector quantization 202-208

W

word lemmatization
performing 138, 139

words
stemming 135-137

word_tokenize function
URL 129
using 129

Z

zip
using 37-39



Thank you for buying Python Data Science Cookbook

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.

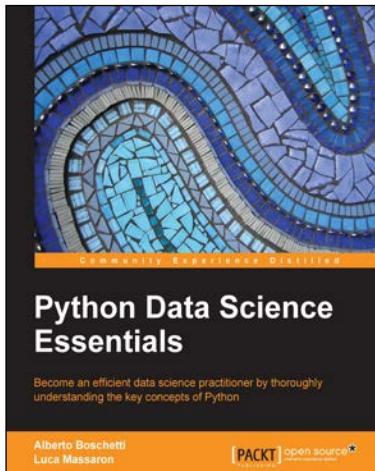
About Packt Open Source

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.

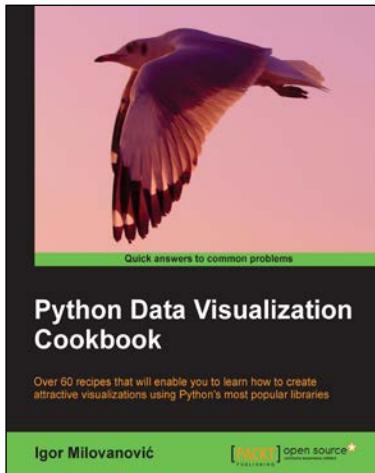


Python Data Science Essentials

ISBN: 978-1-78528-042-9 Paperback: 258 pages

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

1. Quickly get familiar with data science using Python.
2. Save tons of time through this reference book with all the essential tools illustrated and explained.
3. Create effective data science projects and avoid common pitfalls with the help of examples and hints dictated by experience.



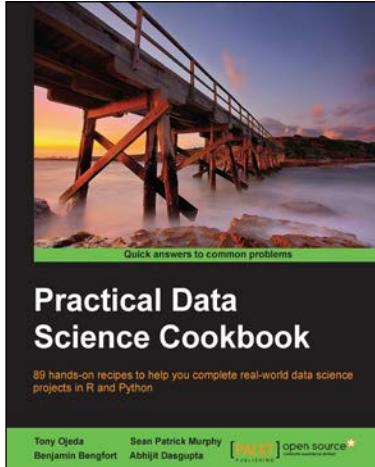
Python Data Visualization Cookbook

ISBN: 978-1-78216-336-7 Paperback: 280 pages

Over 60 recipes that will enable you to learn how to create attractive visualizations using Python's most popular libraries

1. Learn how to set up an optimal Python environment for data visualization.
2. Understand the topics such as importing data for visualization and formatting data for visualization.
3. Understand the underlying data and how to use the right visualizations.

Please check www.PacktPub.com for information on our titles

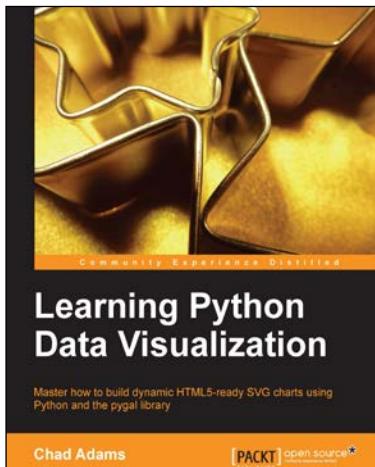


Practical Data Science Cookbook

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.



Learning Python Data Visualization

ISBN: 978-1-78355-333-4 Paperback: 212 pages

Master how to build dynamic HTML5-ready SVG charts using Python and the pygal library

1. A practical guide that helps you break into the world of data visualization with Python.
2. Understand the fundamentals of building charts in Python.
3. Packed with easy-to-understand tutorials for developers who are new to Python or charting in Python.

Please check www.PacktPub.com for information on our titles