

Raúl Garreta, Guillermo Moncecchi,
Trent Hauck, Gavin Hackeling

Scikit-learn: Machine Learning Simplified

Learning Path

Implement scikit-learn into every step of the data science pipeline



Packt

scikit-learn: Machine Learning Simplified

Table of Contents

[Credits](#)

[Preface](#)

[What this learning path covers](#)

[What you need for this learning path](#)

[Who this learning path is for](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Module 1](#)

[1. Machine Learning – A Gentle Introduction](#)

[Installing scikit-learn](#)

[Linux](#)

[Mac](#)

[Windows](#)

[Checking your installation](#)

[Datasets](#)

[Our first machine learning method – linear classification](#)

[Evaluating our results](#)

[Machine learning categories](#)

Important concepts related to machine learning

Summary

2. Supervised Learning

Image recognition with Support Vector Machines

Training a Support Vector Machine

Text classification with Naïve Bayes

Preprocessing the data

Training a Naïve Bayes classifier

Evaluating the performance

Explaining Titanic hypothesis with decision trees

Preprocessing the data

Training a decision tree classifier

Interpreting the decision tree

Random Forests – randomizing decisions

Evaluating the performance

Predicting house prices with regression

First try – a linear model

Second try – Support Vector Machines for regression

Third try – Random Forests revisited

Evaluation

Summary

3. Unsupervised Learning

Principal Component Analysis

Clustering handwritten digits with k-means

Alternative clustering methods

Summary

4. Advanced Features

Feature extraction

Feature selection

Model selection

Grid search

Parallel grid search

Summary

2. Module 2

1. Premodel Workflow

Introduction

Getting sample data from external sources

Getting ready

How to do it...

How it works...

There's more...

See also

Creating sample data for toy analysis

Getting ready

How to do it...

How it works...

Scaling data to the standard normal

Getting ready

How to do it...

How it works...

There's more...

Creating idempotent scalar objects

Handling sparse imputations

Creating binary features through thresholding

Getting ready

How to do it...

How it works...

There's more...

Sparse matrices

The fit method

Working with categorical variables

Getting ready

How to do it...

How it works...

There's more...

DictVectorizer

Patsy

Binarizing label features

Getting ready

How to do it...

How it works...

There's more...

Imputing missing values through various strategies

Getting ready

How to do it...

How it works...

There's more...

Using Pipelines for multiple preprocessing steps

Getting ready

How to do it...

[How it works...](#)

[Reducing dimensionality with PCA](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Using factor analysis for decomposition](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Kernel PCA for nonlinear dimensionality reduction](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Using truncated SVD to reduce dimensionality](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Sign flipping](#)

[Sparse matrices](#)

[Decomposition to classify with DictionaryLearning](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Putting it all together with Pipelines](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Using Gaussian processes for regression](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Defining the Gaussian process object directly](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Using stochastic gradient descent for regression](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[2. Working with Linear Models](#)

[Introduction](#)

[Fitting a line through data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Evaluating the linear regression model](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Using ridge regression to overcome linear regression's shortfalls](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Optimizing the ridge regression parameter](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Using sparsity to regularize models](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Lasso cross-validation](#)

[Lasso for feature selection](#)

[Taking a more fundamental approach to regularization with LARS](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Using linear methods for classification – logistic regression](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Directly applying Bayesian ridge regression](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Using boosting to learn from errors](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[3. Building Models with Distance Metrics](#)

[Introduction](#)

[Using KMeans to cluster data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Optimizing the number of centroids](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Assessing cluster correctness](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Using MiniBatch KMeans to handle more data](#)

[Getting ready](#)

[How to do it...](#)

How it works...

Quantizing an image with KMeans clustering

Getting ready

How do it...

How it works...

Finding the closest objects in the feature space

Getting ready

How to do it...

How it works...

There's more...

Probabilistic clustering with Gaussian Mixture

Models

Getting ready

How to do it...

How it works...

Using KMeans for outlier detection

Getting ready

How to do it...

How it works...

Using k-NN for regression

Getting ready

How to do it...

How it works...

4. Classifying Data with scikit-learn

Introduction

Doing basic classifications with Decision Trees

Getting ready

[How to do it...](#)

[How it works...](#)

[Tuning a Decision Tree model](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Using many Decision Trees – random forests](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Tuning a random forest model](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Classifying data with support vector machines](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Generalizing with multiclass classification](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Using LDA for classification](#)

[Getting ready](#)

How to do it...

How it works...

Working with QDA – a nonlinear LDA

Getting ready

How to do it...

How it works...

Using Stochastic Gradient Descent for classification

Getting ready

How to do it...

Classifying documents with Naïve Bayes

Getting ready

How to do it...

How it works...

There's more...

Label propagation with semi-supervised learning

Getting ready

How to do it...

How it works...

5. Postmodel Workflow

Introduction

K-fold cross validation

Getting ready

How to do it...

How it works...

Automatic cross validation

Getting ready

How to do it...

How it works...

Cross validation with ShuffleSplit

Getting ready

How to do it...

Stratified k-fold

Getting ready

How to do it...

How it works...

Poor man's grid search

Getting ready

How to do it...

How it works...

Brute force grid search

Getting ready

How to do it...

How it works...

Using dummy estimators to compare results

Getting ready

How to do it...

How it works...

Regression model evaluation

Getting ready

How to do it...

How it works...

Feature selection

Getting ready

How to do it...

How it works...

Feature selection on L1 norms

Getting ready

How to do it...

How it works...

Persisting models with joblib

Getting ready

How to do it...

How it works...

There's more...

3. Module 3

1. The Fundamentals of Machine Learning

Learning from experience

Machine learning tasks

Training data and test data

Performance measures, bias, and variance

An introduction to scikit-learn

Installing scikit-learn

Installing scikit-learn on Windows

Installing scikit-learn on Linux

Installing scikit-learn on OS X

Verifying the installation

Installing pandas and matplotlib

Summary

2. Linear Regression

Simple linear regression

Evaluating the fitness of a model with a cost

function

Solving ordinary least squares for simple linear regression

Evaluating the model

Multiple linear regression

Polynomial regression

Regularization

Applying linear regression

Exploring the data

Fitting and evaluating the model

Fitting models with gradient descent

Summary

3. Feature Extraction and Preprocessing

Extracting features from categorical variables

Extracting features from text

The bag-of-words representation

Stop-word filtering

Stemming and lemmatization

Extending bag-of-words with TF-IDF weights

Space-efficient feature vectorizing with the hashing trick

Extracting features from images

Extracting features from pixel intensities

Extracting points of interest as features

SIFT and SURF

Data standardization

Summary

4. From Linear Regression to Logistic Regression

Binary classification with logistic regression

Spam filtering

Binary classification performance metrics

Accuracy

Precision and recall

Calculating the F1 measure

ROC AUC

Tuning models with grid search

Multi-class classification

Multi-class classification performance metrics

Multi-label classification and problem

transformation

Multi-label classification performance metrics

Summary

5. Nonlinear Classification and Regression with Decision Trees

Decision trees

Training decision trees

Selecting the questions

Information gain

Gini impurity

Decision trees with scikit-learn

Tree ensembles

The advantages and disadvantages of decision trees

Summary

6. Clustering with K-Means

Clustering with the K-Means algorithm

Local optima

The elbow method

Evaluating clusters

Image quantization

Clustering to learn features

Summary

7. Dimensionality Reduction with PCA

An overview of PCA

Performing Principal Component Analysis

Variance, Covariance, and Covariance Matrices

Eigenvectors and eigenvalues

Dimensionality reduction with Principal Component Analysis

Using PCA to visualize high-dimensional data

Face recognition with PCA

Summary

8. The Perceptron

Activation functions

The perceptron learning algorithm

Binary classification with the perceptron

Document classification with the perceptron

Limitations of the perceptron

Summary

9. From the Perceptron to Support Vector Machines

Kernels and the kernel trick

[Maximum margin classification and support vectors](#)

[Classifying characters in scikit-learn](#)

[Classifying handwritten digits](#)

[Classifying characters in natural images](#)

[Summary](#)

[10. From the Perceptron to Artificial Neural Networks](#)

[Nonlinear decision boundaries](#)

[Feedforward and feedback artificial neural networks](#)

[Multilayer perceptrons](#)

[Minimizing the cost function](#)

[Forward propagation](#)

[Backpropagation](#)

[Approximating XOR with Multilayer perceptrons](#)

[Classifying handwritten digits](#)

[Summary](#)

[Bibliography](#)

[Index](#)

scikit-learn: Machine Learning Simplified

scikit-learn: Machine Learning Simplified

Copyright © 2017 Packt Publishing

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

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

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

Published on: November 2017

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84719-752-8

www.packtpub.com

Credits

Authors

Raúl Garreta

Guillermo Moncecchi

Trent Hauck

Gavin Hackeling

Reviewers

Andreas Hjortgaard Danielsen

Noel Dawe

Gavin Hackeling

Anoop Thomas Mathew

Xingzhong

Fahad Arshad

Sarah Guido

Mikhail Korobov

Aman Madaan

Content Development Editor

Mayur Pawanikar

Production Coordinator

Arvindkumar Gupta

Preface

Suppose you want to predict whether tomorrow will be a sunny or rainy day. You can develop an algorithm that is based on the current weather and your meteorological knowledge using a rather complicated set of rules to return the desired prediction. Now suppose that you have a record of the day-by-day weather conditions for the last five years, and you find that every time you had two sunny days in a row, the following day also happened to be a sunny one. Your algorithm could generalize this and predict that tomorrow will be a sunny day since the sun reigned today and yesterday. This algorithm is a pretty simple example of learning from experience. This is what Machine Learning is all about: algorithms that learn from the available data.

This course is designed in the same way that many data science and analytics projects play out. First, we need to acquire data; the data is often messy, incomplete, or not correct in some way. Therefore, we spend the first chapter talking about strategies for dealing with bad data and ways to deal with other problems that arise from data. For example, what happens if we have too many features? How do we handle that?

What this learning path covers

Module 1, Learning scikit-learn: Machine Learning in Python, in this module, you will learn several methods for building Machine Learning applications that solve different real-world tasks, from document classification to image recognition. We will use Python, a simple, popular, and widely used programming language, and scikit-learn, an open source Machine Learning library. In each chapter of this module, we will present a different Machine Learning setting and a couple of well-studied methods as well as show step-by-step examples that use Python and scikit-learn to solve concrete tasks. We will also show you tips and tricks to improve algorithm performance, both from the accuracy and computational cost point of views.

Module 2, scikit-learn Cookbook, the first chapter of this module is your guide. The meat of this module will walk you through various algorithms and how to implement them into your workflow. And finally, we'll end with the postmodel workflow. This chapter is fairly agnostic to the other chapters of the module and can be applied to the various algorithms you'll learn up until the final chapter.

Module 3, Mastering Machine Learning with scikit-learn, in this module, we will examine several machine learning models and learning algorithms. We will discuss tasks that machine learning is commonly applied to, and learn to measure the performance of machine learning systems. We will work with a popular library for the Python programming language called scikit-learn, which has assembled excellent implementations of many machine learning models and algorithms under a simple yet versatile API.

This module is motivated by two goals:

- Its content should be accessible. The book only assumes familiarity with basic programming and math.
- Its content should be practical. This book offers hands-on examples that readers can adapt to problems in the real world.

What you need for this learning path

Module 1:

For running the module's examples, you will need a running Python environment, including the scikit-learn library and NumPy and SciPy mathematical libraries. The source code will be available in the form of IPython notebooks. For Chapter 4, Advanced Features, we will also include the Pandas Python library. Chapter 1, Machine Learning – A Gentle Introduction, shows how to install them in your operating system.

Module 2:

Here are the contents that will get the environment set up. This will allow you to follow along with the code in this module. This method may be easier for less-experienced Python developers:

dateutil==2.1

ipython==2.2.0

ipython-notebook==2.1.0

jinja2==2.7.3

markupsafe==0.18

matplotlib==1.3.1

numpy==1.8.1

patsy==0.3.0

pandas==0.14.1

pip==1.5.6

pydot==1.0.28

pyparsing==1.5.6

pytz==2014.4

pyzmq==14.3.1

scikit-learn==0.15.0

scipy==0.14.0

setuptools==3.6

six==1.7.3

ssl_match_hostname==3.4.0.2

tornado==3.2.2

Module 3:

The examples in this module assume that you have an installation of Python 2.7. The first chapter will describe methods to install scikit-learn 0.15.2, its dependencies, and other libraries on Linux, OS X, and Windows.

Who this learning path is for

If you are a programmer and want to explore machine learning and data-based methods to build intelligent applications and enhance your programming skills, this is the book for you. No previous experience with machine-learning algorithms is required.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

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

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

need to be logged in to your Packt account.

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

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

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/scikit-learn-Machine-Learning-Simplified>. We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

Errata

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

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

Piracy

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

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

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

Questions

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

Part 1. Module 1

Learning scikit-learn: Machine Learning in Python

Experience the benefits of machine learning techniques by applying them to real-world problems using Python and the open source scikit-learn library

Chapter 1. Machine Learning – A Gentle Introduction

"I was into data before it was big"—@ml_hipster

You have probably heard recently about big data. The Internet, the explosion of electronic devices with tremendous computational power, and the fact that almost every process in our world uses some kind of software, are giving us huge amounts of data every minute.

Think about social networks, where we store information about people, their interests, and their interactions. Think about process-control devices, ranging from web servers to cars and pacemakers, which permanently leave logs of data about their performance. Think about scientific research initiatives, such as the genome project, which have to analyze huge amounts of data about our DNA.

There are many things you can do with this data: examine it, summarize it, and even visualize it in several beautiful ways. However, this book deals with another use for data: as a source of experience to improve our algorithms'

performance. These algorithms, which can learn from previous data, conform to the field of Machine Learning, a subfield of Artificial Intelligence.

Any machine learning problem can be represented with the following three concepts:

- We will have to learn to solve a task T . For example, build a spam filter that learns to classify e-mails as spam or ham.
- We will need some experience E to learn to perform the task. Usually, experience is represented through a dataset. For the spam filter, experience comes as a set of e-mails, manually classified by a human as spam or ham.
- We will need a measure of performance P to know how well we are solving the task and also to know whether after doing some modifications, our results are improving or getting worse. The percentage of e-mails that our spam filtering is correctly classifying as spam or ham could be P for our spam-filtering task.

Scikit-learn is an open source Python library of popular machine learning algorithms that will allow us to build these types of systems. The project was started in 2007 as a *Google Summer of Code* project by *David Cournapeau*.

Later that year, *Matthieu Brucher* started working on this project as part of his thesis. In 2010, *Fabian Pedregosa*, *Gael Varoquaux*, *Alexandre Gramfort*, and *Vincent Michel* of INRIA took the project leadership and produced the first public release. Nowadays, the project is being developed very actively by an enthusiastic community of contributors. It is built upon NumPy (<http://www.numpy.org/>) and SciPy (<http://scipy.org/>), the standard Python libraries for scientific computation. Through this book, we will use it to show you how the incorporation of previous data as a source of experience could serve to solve several common programming tasks in an efficient and probably more effective way.

In the following sections of this chapter, we will start viewing how to install scikit-learn and prepare your working environment. After that, we will have a brief introduction to machine learning in a practical way, trying to introduce key machine learning concepts while solving a simple practical task.

Installing scikit-learn

Installation instructions for scikit-learn are available at <http://scikit-learn.org/stable/install.html>. Several examples in this book include visualizations, so you should also install the `matplotlib` package from <http://matplotlib.org/>. We also recommend installing IPython Notebook, a very useful tool that includes a web-based console to edit and run code snippets, and render the results. The source code that comes with this book is provided through IPython notebooks.

An easy way to install all packages is to download and install the Anaconda distribution for scientific computing from <https://store.continuum.io/>, which provides all the necessary packages for Linux, Mac, and Windows platforms. Or, if you prefer, the following sections gives some suggestions on how to install every package on each particular platform.

Linux

Probably the easiest way to install our environment is through the operating system packages. In the case of Debian-based operating systems, such as Ubuntu, you can install the packages by running the following commands:

- Firstly, to install the package we enter the following command:

```
sudo apt-get install build-essential python-dev python-numpy python-setuptools python-scipy libatlas-dev python-pip
```

- Then, to install matplotlib, run the following command:

```
# sudo apt-get install python-matplotlib
```

- After that, we should be ready to install scikit-learn by issuing this command:

```
# sudo pip install scikit-learn
```

- To install IPython Notebook, run the following command:

```
# sudo apt-get install ipython-notebook
```

- If you want to install from source, let's say to install all the libraries within a virtual environment, you should issue the following commands:

```
# pip install numpy  
# pip install scipy  
# pip install scikit-learn
```

- To install Matplotlib, you should run the following commands:

```
# pip install libpng-dev libjpeg8-dev  
libfreetype6-dev  
# pip install matplotlib
```

- To install IPython Notebook, you should run the following commands:

```
# pip install ipython  
# pip install tornado  
# pip install pyzmq
```

Mac

You can similarly use tools such as MacPorts and HomeBrew that contain precompiled versions of these packages.

Windows

To install scikit-learn on Windows, you can download a Windows installer from the downloads section of the project web page: <http://sourceforge.net/projects/scikit-learn/files/>

Checking your installation

To check that everything is ready to run, just open your Python (or probably better, IPython) console and type the following:

```
>>> import sklearn as sk  
>>> import numpy as np  
>>> import matplotlib.pyplot as plt
```

We have decided to precede Python code with `>>>` to separate it from the sentence results. Python will silently import the scikit-learn, NumPy, and matplotlib packages, which we will use through the rest of this book's examples.

If you want to execute the code presented in this book, you should run IPython Notebook:

```
# ipython notebook
```

This will allow you to open the corresponding notebooks right in your browser.

Datasets

As we have said, machine learning methods rely on previous experience, usually represented by a dataset. Every method implemented on scikit-learn assumes that data comes in a dataset, a certain form of input data representation that makes it easier for the programmer to try different methods on the same data. Scikit-learn includes a few well-known datasets. In this chapter, we will use one of them, the Iris flower dataset, introduced in 1936 by *Sir Ronald Fisher* to show how a statistical method (discriminant analysis) worked (yes, they were into data before it was big). You can find a description of this dataset on its own Wikipedia page, but, essentially, it includes information about 150 elements (or, in machine learning terminology, instances) from three different Iris flower species, including sepal and petal length and width. The natural task to solve using this dataset is to learn to guess the Iris species knowing the sepal and petal measures. It has been widely used on machine learning tasks because it is a very easy dataset in a sense that we will see later. Let's import the dataset and show the values for the first instance:

```
>>> from sklearn import datasets  
>>> iris = datasets.load_iris()  
>>> X_iris, y_iris = iris.data, iris.target
```

```
>>> print X_iris.shape, y_iris.shape  
(150, 4) (150,)  
>>> print X_iris[0], y_iris[0]  
[ 5.1  3.5  1.4  0.2] 0
```

Tip

Downloading the example code

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

We can see that the `iris` dataset is an object (similar to a dictionary) that has two main components:

- A `data array`, where, for each instance, we have the real values for sepal length, sepal width, petal length, and petal width, in that order (note that for efficiency reasons, scikit-learn methods work on NumPy `ndarrays` instead of the more descriptive but much less efficient Python dictionaries or lists). The shape of this array is `(150, 4)`, meaning that we have 150 rows (one for each instance) and four columns (one for each feature).
- A `targetarray`, with values in the range of 0 to 2,

corresponding to each instance of Iris species (0: setosa, 1: versicolor, and 2: virginica), as you can verify by printing the `iris.target_names` value.

While it's not necessary for every dataset we want to use with scikit-learn to have this exact structure, we will see that every method will require this data array, where each instance is represented as a list of features or attributes, and another target array representing a certain value we want our learning method to learn to predict. In our example, the petal and sepal measures are our real-valued attributes, while the flower species is the one-of-a-list class we want to predict.

Our first machine learning method –linear classification

To get a grip on the problem of machine learning in scikit-learn, we will start with a very simple machine learning problem: we will try to predict the Iris flower species using only two attributes: sepal width and sepal length. This is an instance of a classification problem, where we want to assign a label (a value taken from a discrete set) to an item according to its features.

Let's first build our training dataset—a subset of the original sample, represented by the two attributes we selected and their respective target values. After importing the dataset, we will randomly select about 75 percent of the instances, and reserve the remaining ones (the evaluation dataset) for evaluation purposes (we will see later why we should always do that):

```
>>> from sklearn.cross_validation import  
train_test_split  
>>> from sklearn import preprocessing  
>>> # Get dataset with only the first two  
attributes
```

```
>>> X, y = X_iris[:, :2], y_iris
>>> # Split the dataset into a training and a
testing set
>>> # Test set will be the 25% taken randomly
>>> X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.25,
random_state=33)
>>> print X_train.shape, y_train.shape
(112, 2) (112,)
>>> # Standardize the features
>>> scaler =
preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
```

The `train_test_split` function automatically builds the training and evaluation datasets, randomly selecting the samples. Why not just select the first 112 examples? This is because it could happen that the instance ordering within the sample could matter and that the first instances could be different to the last ones. In fact, if you look at the Iris datasets, the instances are ordered by their target class, and this implies that the proportion of 0 and 1 classes will be higher in the new training set, compared with that of the original dataset. We always want our training data to be a representative sample of the population they represent.

The last three lines of the previous code modify the training set in a process usually called feature scaling. For

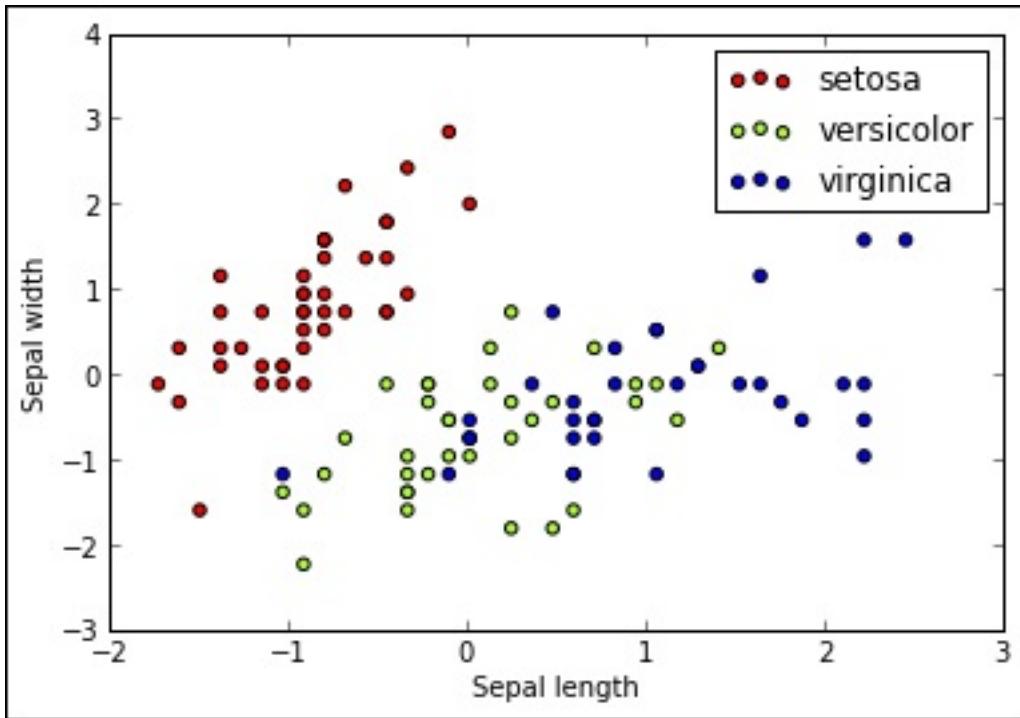
each feature, calculate the average, subtract the mean value from the feature value, and divide the result by their standard deviation. After scaling, each feature will have a zero average, with a standard deviation of one. This standardization of values (which does not change their distribution, as you could verify by plotting the x values before and after scaling) is a common requirement of machine learning methods, to avoid that features with large values may weight too much on the final results.

Now, let's take a look at how our training instances are distributed in the two-dimensional space generated by the learning feature. `pyplot`, from the `matplotlib` library, will help us with this:

```
>>> import matplotlib.pyplot as plt
>>> colors = ['red', 'greenyellow', 'blue']
>>> for i in xrange(len(colors)):
>>>     xs = x_train[:, 0][y_train == i]
>>>     ys = x_train[:, 1][y_train == i]
>>>     plt.scatter(xs, ys, c=colors[i])
>>> plt.legend(iris.target_names)
>>> plt.xlabel('Sepal length')
>>> plt.ylabel('Sepal width')
```

The `scatter` function simply plots the first feature value (sepal width) for each instance versus its second feature value (sepal length) and uses the target class values to assign a different color for each class. This way, we can

have a pretty good idea of how these attributes contribute to determine the target class. The following screenshot shows the resulting plot:



Looking at the preceding screenshot, we can see that the separation between the red dots (corresponding to the Iris setosa) and green and blue dots (corresponding to the two other Iris species) is quite clear, while separating green from blue dots seems a very difficult task, given the two features available. This is a very common scenario: one of the first questions we want to answer in a machine learning task is if the feature set we are using is actually useful for the task we are solving, or if we need to add

new attributes or change our method.

Given the available data, let's, for a moment, redefine our learning task: suppose we aim, given an Iris flower instance, to predict if it is a setosa or not. We have converted our problem into a binary classification task (that is, we only have two possible target classes).

If we look at the picture, it seems that we could draw a straight line that correctly separates both the sets (perhaps with the exception of one or two dots, which could lie in the incorrect side of the line). This is exactly what our first classification method, linear classification models, tries to do: build a line (or, more generally, a hyperplane in the feature space) that best separates both the target classes, and use it as a decision boundary (that is, the class membership depends on what side of the hyperplane the instance is).

To implement linear classification, we will use the `SGDClassifier` from scikit-learn. **SGD** stands for **Stochastic Gradient Descent**, a very popular numerical procedure to find the local minimum of a function (in this case, the loss function, which measures how far every instance is from our boundary). The algorithm will learn the coefficients of the hyperplane by minimizing the loss function.

To use any method in scikit-learn, we must first create the corresponding classifier object, initialize its parameters, and train the model that better fits the training data. You will see while you advance in this book that this procedure will be pretty much the same for what initially seemed very different tasks.

```
>>>  
from sklearn.linear_model import SGDClassifier  
>>> clf = SGDClassifier()  
>>> clf.fit(X_train, y_train)
```

The `SGDClassifier` initialization function allows several parameters. For the moment, we will use the default values, but keep in mind that these parameters could be very important, especially when you face more real-world tasks, where the number of instances (or even the number of attributes) could be very large. The `fit` function is probably the most important one in scikit-learn. It receives the training data and the training classes, and builds the classifier. Every supervised learning method in scikit-learn implements this function.

What does the classifier look like in our linear model method? As we have already said, every future classification decision depends just on a hyperplane. That hyperplane is, then, our model. The `coef_` attribute of the `clf` object (consider, for the moment, only the first row of

the matrices), now has the coefficients of the linear boundary and the `intercept_` attribute, the point of intersection of the line with the y axis. Let's print them:

```
>>> print clf.coef_
[[ -28.53692691  15.05517618]
 [ -8.93789454 -8.13185613]
 [ 14.02830747 -12.80739966]]
>>> print clf.intercept_
[-17.62477802 -2.35658325 -9.7570213 ]
```

Indeed in the real plane, with these three values, we can draw a line, represented by the following equation:

$$-17.62477802 - 28.53692691 * x_1 + 15.05517618 * x_2 = 0$$

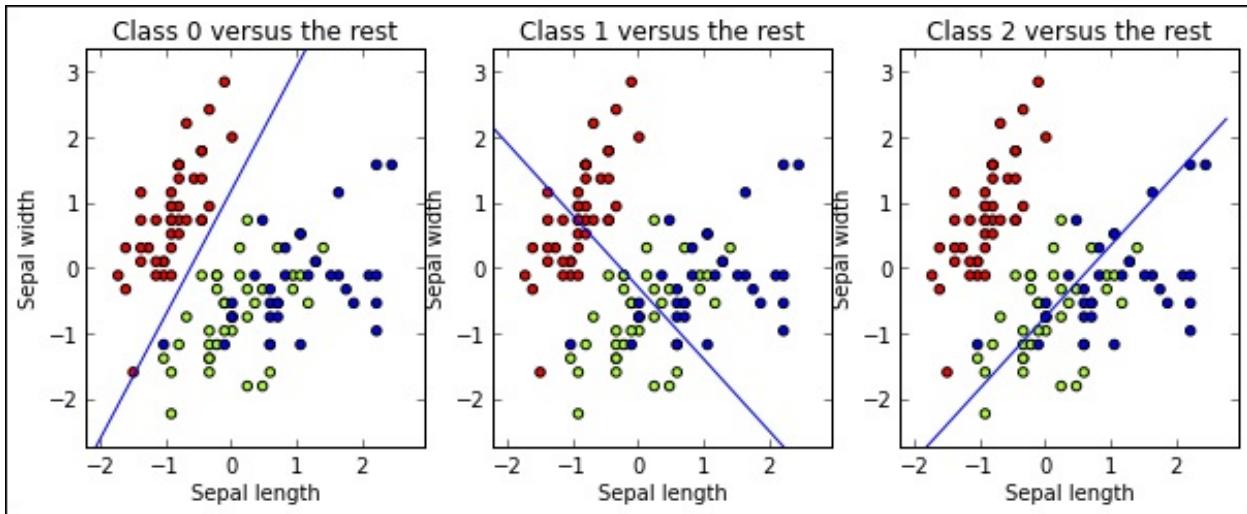
Now, given x_1 and x_2 (our real-valued features), we just have to compute the value of the left-side of the equation: if its value is greater than zero, then the point is above the decision boundary (the red side), otherwise it will be beneath the line (the green or blue side). Our prediction algorithm will simply check this and predict the corresponding class for any new iris flower.

But, why does our coefficient matrix have three rows? Because we did not tell the method that we have changed our problem definition (how could we have done this?), and it is facing a three-class problem, not a binary

decision problem. What, in this case, the classifier does is the same we did—it converts the problem into three binary classification problems in a one-versus-all setting (it proposes three lines that separate a class from the rest).

The following code draws the three decision boundaries and lets us know if they worked as expected:

```
>>> x_min, x_max = X_train[:, 0].min() - .5,
X_train[:, 0].max() +
    .5
>>> y_min, y_max = X_train[:, 1].min() - .5,
X_train[:, 1].max() +
    .5
>>> xs = np.arange(x_min, x_max, 0.5)
>>> fig, axes = plt.subplots(1, 3)
>>> fig.set_size_inches(10, 6)
>>> for i in [0, 1, 2]:
>>>     axes[i].set_aspect('equal')
>>>     axes[i].set_title('Class ' + str(i) + ' '
versus the rest')
>>>     axes[i].set_xlabel('Sepal length')
>>>     axes[i].set_ylabel('Sepal width')
>>>     axes[i].set_xlim(x_min, x_max)
>>>     axes[i].set_ylim(y_min, y_max)
>>>     pylab.sca(axes[i])
>>>     plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
                    cmap=plt.cm.prism)
>>>     ys = (-clf.intercept_[i] -
            xs * clf.coef_[i, 0]) / clf.coef_[i, 1]
>>>     plt.plot(xs, ys, hold=True)
```



The first plot shows the model built for our original binary problem. It looks like the line separates quite well the Iris setosa from the rest. For the other two tasks, as we expected, there are several points that lie on the wrong side of the hyperplane.

Now, the end of the story: suppose that we have a new flower with a sepal width of 4.7 and a sepal length of 3.1, and we want to predict its class. We just have to apply our brand new classifier to it (after normalizing!). The predict method takes an array of instances (in this case, with just one element) and returns a list of predicted classes:

```
>>>print clf.predict(scaler.transform([[4.7,
3.1]]))
[0]
```

If our classifier is right, this Iris flower is a setosa. Probably, you have noticed that we are predicting a class from the possible three classes but that linear models are essentially binary: something is missing. You are right. Our prediction procedure combines the result of the three binary classifiers and selects the class in which it is more confident. In this case, we will select the boundary line whose distance to the instance is longer. We can check that using the classifier `decision_function` method:

```
>>>print  
clf.decision_function(scaler.transform([[4.7,  
3.1]]))  
[[ 19.73905808    8.13288449   -28.63499119]]
```

Evaluating our results

We want to be a little more formal when we talk about a good classifier. What does that mean? The performance of a classifier is a measure of its effectiveness. The simplest performance measure is accuracy: given a classifier and an evaluation dataset, it measures the proportion of instances correctly classified by the classifier. First, let's test the accuracy on the training set:

```
>>> from sklearn import metrics  
>>> y_train_pred = clf.predict(X_train)  
>>> print metrics.accuracy_score(y_train,  
y_train_pred)  
0.821428571429
```

This figure tells us that 82 percent of the training set instances are correctly classified by our classifier.

Probably, the most important thing you should learn from this chapter is that measuring accuracy on the training set is really a bad idea. You have built your model using this data, and it is possible that your model adjusts well to them but performs poorly in future (previously unseen data), which is its purpose. This phenomenon is called **overfitting**, and you will see it now and again while you read this book. If you measure based on your training

data, you will never detect overfitting. So, never measure based on your training data.

This is why we have reserved part of the original dataset (the testing partition)—we want to evaluate performance on previously unseen data. Let's check the accuracy again, now on the evaluation set (recall that it was already scaled):

```
>>> y_pred = clf.predict(x_test)
>>> print metrics.accuracy_score(y_test, y_pred)
0.684210526316
```

We obtained an accuracy of 68 percent in our testing set. Usually, accuracy on the testing set is lower than the accuracy on the training set, since the model is actually modeling the training set, not the testing set. Our goal will always be to produce models that avoid overfitting when trained over a training set, so they have enough generalization power to also correctly model the unseen data.

Accuracy on the test set is a good performance measure when the number of instances of each class is similar, that is, we have a uniform distribution of classes. But if you have a skewed distribution (say, 99 percent of the instances belong to one class), a classifier that always predicts the majority class could have an excellent

performance in terms of accuracy despite the fact that it is an extremely naive method.

Within scikit-learn, there are several evaluation functions; we will show three popular ones: precision, recall, and F1-score (or f-measure). They assume a binary classification problem and two classes—a positive one and a negative one. In our example, the positive class could be Iris setosa, while the other two will be combined into one negative class.

- **Precision:** This computes the proportion of instances predicted as positives that were correctly evaluated (it measures how right our classifier is when it says that an instance is positive).
- **Recall:** This counts the proportion of positive instances that were correctly evaluated (measuring how right our classifier is when faced with a positive instance).
- **F1-score:** This is the harmonic mean of precision and recall, and tries to combine both in a single number.

Note

The harmonic mean is used instead of the arithmetic mean because the latter compensates low values for precision and high values for recall (and vice versa). On the

other hand, with harmonic mean we will always have low values if either precision or recall is low. For an interesting description of this issue refer to the paper <http://www.cs.odu.edu/~mukka/cs795sum12dm/Lecture no measure-YS-26Oct07.pdf>

We can define these measures in terms of True and False, and Positives and Negatives:

	Prediction: Positive	Prediction: Negative
Target cass: Positive	True Positive (TP)	False Negative (FN)
Target cass: Negative	False Positive (FP)	True Negative (TN)

With m being the sample size (that is, $TP + TN + FP + FN$), we have the following formulae:

- Accuracy = $(TP + TN) / m$
- Precision = $TP / (TP + FP)$
- Recall = $TP / (TP + FN)$
- F1-score = $2 * \text{Precision} * \text{Recall} / (\text{Precision} + \text{Recall})$

Let's see it in practice:

```
>>> print metrics.classification_report(y_test,  
y_pred,target_names=iris.target_names)
```

	precision	recall	f1-score
support			
setosa	1.00	1.00	1.00
8			
versicolor	0.43	0.27	0.33
11			
virginica	0.65	0.79	0.71
19			
avg / total	0.66	0.68	0.66
38			

We have computed precision, recall, and f1-score for each class and their average values. What we can see in this table is:

- The classifier obtained 1.0 precision and recall in the `setosa` class. This means that for precision, 100 percent of the instances that are classified as setosa are really setosa instances, and for recall, that 100 percent of the setosa instances were classified as setosa.
- On the other hand, in the `versicolor` class, the results are not as good: we have a precision of 0.43, that is, only 43 percent of the instances that are classified as versicolor are really versicolor instances. Also, for versicolor, we have a recall of 0.27, that is, only 27 percent of the versicolor instances are correctly classified.

Now, we can see that our method (as we expected) is very good at predicting `setosa`, while it suffers when it has to separate the `versicolor` or `virginica` classes. The support value shows how many instances of each class we had in the testing set.

Another useful metric (especially for multi-class problems) is the confusion matrix: in its (i, j) cell, it shows the number of class instances i that were predicted to be in class j . A good classifier will accumulate the values on the confusion matrix diagonal, where correctly classified instances belong.

```
>>> print metrics.confusion_matrix(y_test,  
y_pred)  
[[ 8  0  0]  
 [ 0  3  8]  
 [ 0  4 15]]
```

Our classifier is never wrong in our evaluation set when it classifies class 0 (`setosa`) flowers. But, when it faces classes 1 and 2 flowers (`versicolor` and `virginica`), it confuses them. The confusion matrix gives us useful information to know what types of errors the classifier is making.

To finish our evaluation process, we will introduce a very useful method known as cross-validation. As we

explained before, we have to partition our dataset into a training set and a testing set. However, partitioning the data, results such that there are fewer instances to train on, and also, depending on the particular partition we make (usually made randomly), we can get either better or worse results. Cross-validation allows us to avoid this particular case, reducing result variance and producing a more realistic score for our models. The usual steps for k -fold cross-validation are the following:

1. Partition the dataset into k different subsets.
2. Create k different models by training on $k-1$ subsets and testing on the remaining subset.
3. Measure the performance on each of the k models and take the average measure.

Let's do that with our linear classifier. First, we will have to create a composite estimator made by a pipeline of the standardization and linear models. With this technique, we make sure that each iteration will standardize the data and then train/test on the transformed data. The `Pipeline` class is also useful to simplify the construction of more complex models that chain-multiply the transformations. We will chose to have $k = 5$ folds, so each time we will train on 80 percent of the data and test on the remaining 20 percent. Cross-validation, by default, uses accuracy as its performance measure, but we could select the

measurement by passing any scorer function as an argument.

```
>>> from sklearn.cross_validation import  
cross_val_score, KFold  
>>> from sklearn.pipeline import Pipeline  
>>> # create a composite estimator made by a  
pipeline of the  
    standarization and the linear model  
clf = Pipeline([  
    ('scaler',  
preprocessing.StandardScaler()),  
    ('linear_model', SGDClassifier())  
)  
>>> # create a k-fold cross validation iterator  
of k=5 folds  
>>> cv = KFold(X.shape[0], 5, shuffle=True,  
random_state=33)  
>>> # by default the score used is the one  
returned by score  
    method of the estimator (accuracy)  
>>> scores = cross_val_score(clf, X, y, cv=cv)  
>>> print scores  
[ 0.66666667  0.93333333  0.66666667  0.7  
0.6          ]
```

We obtained an array with the k scores. We can calculate the mean and the standard error to obtain a final figure:

```
>>> from scipy.stats import sem  
>>> def mean_score(scores):  
    return ("Mean score: {:.3f} (+/-  
    {:.3f})").format(np.mean(scores),
```

```
sem(scores))  
>>> print mean_score(scores)  
Mean score: 0.713 (+/-0.057)
```

Our model has an average accuracy of 0.71.

Machine learning categories

Classification is only one of the possible machine learning problems that can be addressed with scikit-learn. We can organize them in the following categories:

- In the previous example, we had a set of instances (that is, a set of data collected from a population) represented by certain features and with a particular target attribute. Supervised learning algorithms try to build a model from this data, which lets us predict the target attribute for new instances, knowing only these instance features. When the target class belongs to a discrete set (such as a list of flower species), we are facing a classification problem.
- Sometimes the class we want to predict, instead of belonging to a discrete set, ranges on a continuous set, such as the real number line. In this case, we are trying to solve a **regression** problem (the term was coined by Francis Galton, who observed that the heights of tall ancestors tend to regress down towards a normal value, the average human height). For example, we could try to predict the petal width based on the other three features. We will see that the

methods used for regression are quite different from those used for classification.

- Another different type of machine learning problem is that of **unsupervised learning**. In this case, we do not have a target class to predict but instead want to group instances according to some similarity measure based on the available set of features. For example, suppose you have a dataset composed of e-mails and want to group them by their main topic (the task of grouping instances is called **clustering**). We can use it as features, for example, the different words used in each of them.

Important concepts related to machine learning

The linear classifier we presented in the previous section could look too simple. What if we use a higher degree polynomial? What if we also take as features not only the sepal length and width, but also the petal length and the petal width? This is perfectly possible, and depending on the sample distribution, it could lead to a better fit to the training data, resulting in higher accuracy. The problem with this approach is that now we must estimate not only the three original parameters (the coefficients for x_1 , x_2 , and the interception point), but also the parameters for the new features x_3 and x_4 (petal length and width) and also the product combinations of the four features.

Intuitively, we would need more training data to adequately estimate these parameters. The number of parameters (and consequently, the amount of training data needed to adequately estimate them) would rapidly grow if we add more features or higher order terms. This phenomenon, present in every machine learning method, is called the idem curse of dimensionality: when the number of parameters of a model grows, the data needed to learn them grows exponentially.

This notion is closely related to the problem of overfitting mentioned earlier. As our training data is not enough, we risk producing a model that could be very good at predicting the target class on the training dataset but fail miserably when faced with new data, that is, our model does not have the generalization power. That is why it is so important to evaluate our methods on previously unseen data.

The general rule is that, in order to avoid overfitting, we should prefer simple (that is, with less parameters) methods, something that could be seen as an instantiation of the philosophical principle of Occam's razor, which states that among competing hypotheses, the hypothesis with the fewest assumptions should be selected.

However, we should also take into account Einstein's words:

"Everything should be made as simple as possible, but not simpler."

The idem curse of dimensionality may suggest that we keep our models simple, but on the other hand, if our model is too simple we run the risk of suffering from underfitting. Underfitting problems arise when our model has such a low representation power that it cannot model

the data even if we had all the training data we want. We clearly have underfitting when our algorithm cannot achieve good performance measures even when measuring on the training set.

As a result, we will have to achieve a balance between overfitting and underfitting. This is one of the most important problems that we will have to address when designing our machine learning models.

Other key concepts to take into account are the idem bias and variance of a machine learning method. Consider an extreme method that, in a binary classification setting, always predicts the positive class for any new instance. Its predictions are, trivially, always the same, or in statistical terms, it has null variance; but it will fail to predict negative examples: it is very biased towards positive results. On the other hand, consider a method that predicts, for a new instance, the class of the nearest instance in the training set (in fact, this method exists, and it is called the 1-nearest neighbor). The generalization assumptions that this method uses are very small: it has a very low bias; but, if we change the training data, results could dramatically change, that is, its variance is very high. These are extreme examples of the **bias-variance tradeoff**. It can be shown that, no matter which method we are using, if we reduce bias, variance will increase,

and vice versa.

Linear classifiers have generally low-variance: no matter what subset we select for training, results will be similar. However, if the data distribution (as in the case of the versicolor and virginica species) makes target classes not separable by a hyperplane, these results will be consistently wrong, that is, the method is highly biased.

On the other hand, kNN (a memory-based method we will not address in this book) has very low bias but high variance: the results are generally very good at describing training data but tend to vary greatly when trained on different training instances.

There are other important concepts related to real-world applications where our data will not come naturally as a list of real-valued features. In these cases, we will need to have methods to transform non real-valued features to real-valued ones. Besides, there are other steps related to feature standardization and normalization, which as we saw in our Iris example, are needed to avoid undesired effects regarding the different value ranges. These transformations on the feature space are known as **data preprocessing**.

After having a defined feature set, we will see that not all

of the features that come in our original dataset could be useful for resolving our task. So we must also have methods to do feature selection, that is, methods to select the most promising features.

In this book, we will present several problems and in each of them we will show different ways to transform and find the most relevant features to use for learning a task, called **feature engineering**, which is based on our knowledge of the domain of the problem and/or data analysis methods. These methods, often not valued enough, are a fundamental step toward obtaining good results.

Summary

In this chapter, we introduced the main general concepts in machine learning and presented scikit-learn, the Python library we will use in the rest of this book. We included a very simple example of classification, trying to show the main steps for learning, and including the most important evaluation measures we will use. In the rest of this book, we plan to show you different machine learning methods and techniques using different real-world examples for each one. In almost every computational task, the presence of historical data could allow us to improve performance in the sense introduced at the beginning of this chapter.

The next chapter introduces supervised learning methods: we have annotated data (that is, instances where the target class/value is known) and we want to predict the same class/value for future data from the same population. In the case of classification tasks, that is, a discrete-valued target class, several different models exist, ranging from statistical methods, such as the simple **Naïve Bayes** to advanced linear classifiers, such as **Support Vector Machines (SVM)**. Some methods, such as **decision trees**, will allow us to visualize how important a feature is to discriminate between different target classes and have a

human interpretation of the decision process. We will also address another type of supervised learning task: regression, that is, methods that try to predict real-valued data.

Chapter 2. Supervised Learning

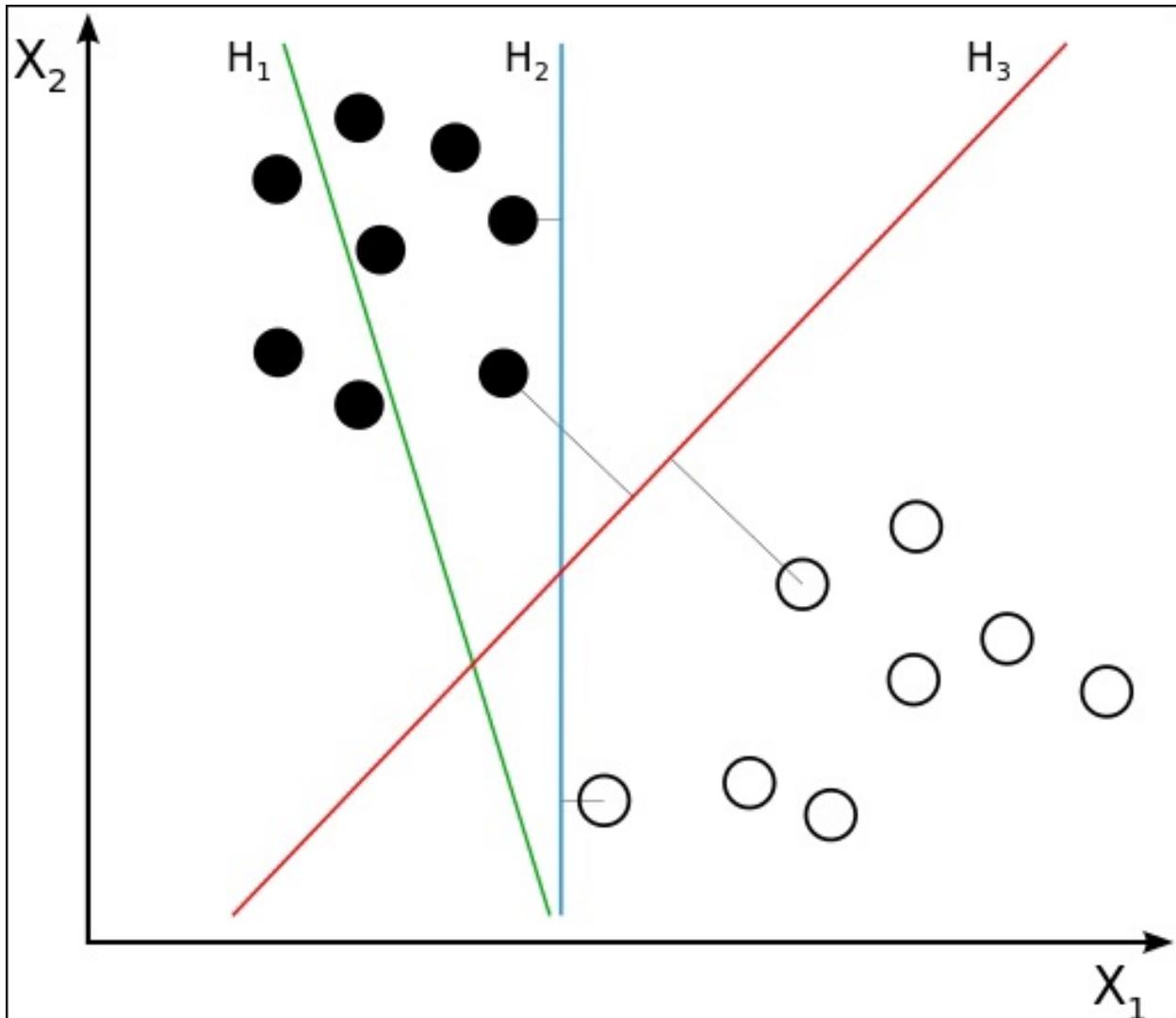
In [Chapter 1](#), *Machine Learning – A Gentle Introduction*, we sketched the general idea of a supervised learning algorithm. We have the training data where each instance has an input (a set of attributes) and a desired output (a target class). Then we use this data to train a model that will predict the same target class for new unseen instances.

Supervised learning methods are nowadays a standard tool in a wide range of disciplines, from medical diagnosis to natural language processing, image recognition, and searching for new particles at the **Large Hadron Collider (LHC)**. In this chapter we will present several methods applied to several real-world examples by using some of the many algorithms implemented in **scikit-learn**. This chapter does not intend to substitute the scikit-learn reference, but is an introduction to the main supervised learning techniques and shows how they can be used to solve practical problems.

Image recognition with Support Vector Machines

Imagine that the instances in your dataset are points in a multidimensional space; we can assume that the model built by our classifier can be a surface or using linear algebra terminology, a hyperplane that separates instances (points) of one class from the rest. **Support Vector Machines (SVM)** are supervised learning methods that try to obtain these hyperplanes in an optimal way, by selecting the ones that pass through the widest possible gaps between instances of different classes. New instances will be classified as belonging to a certain category based on which side of the surfaces they fall on.

The following figure shows an example for a two-dimensional space with two features (**X1** and **X2**) and two classes (black and white):



We can observe that the green hyperplane does not separate both classes, committing some classification errors. The blue and the red hyperplanes separate both classes without errors. However, the red surface separates both classes with maximum margin; it is the most distant hyperplane from the closest instances from the two

categories. The main advantage of this approach is that it will probably lower the generalization error, making this model resistant to overfitting, something that actually has been verified in several, different, classification tasks.

This approach can be generalized to construct hyperplanes not only in two dimensions, but also in high or infinite dimensional spaces. What is more, we can use nonlinear surfaces, such as polynomial or radial basis functions, by using the so called kernel trick, implicitly mapping inputs into high-dimensional feature spaces.

SVM has become one of the state-of-the-art machine learning models for many tasks with excellent results in many practical applications. One of the greatest advantages of SVM is that they are very effective when working on high-dimensional spaces, that is, on problems which have a lot of features to learn from. They are also very effective when the data is sparse (think about a high-dimensional space with very few instances). Besides, they are very efficient in terms of memory storage, since only a subset of the points in the learning space is used to represent the decision surfaces.

To mention some disadvantages, SVM models could be very calculation intensive while training the model and they do not return a numerical indicator of how confident

they are about a prediction. However, we can use some techniques such as K-fold cross-validation to avoid this, at the cost of increasing the computational cost.

We will apply SVM to image recognition, a classic problem with a very large dimensional space (the value of each pixel of the image is considered as a feature). What we will try to do is, given an image of a person's face, predict to which of the possible people from a list does it belongs (this kind of approach is used, for example, in social network applications to automatically tag people within photographs). Our learning set will be a group of labeled images of peoples' faces, and we will try to learn a model that can predict the label of unseen instances. The intuitive and first approach would be to use the image pixels as features for the learning algorithm, so pixel values will be our learning attributes and the individual's label will be our target class.

Our dataset is provided within scikit-learn, so let's start by importing and printing its description.

```
>>> import sklearn as sk
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.datasets import
fetch_olivetti_faces
>>> faces = fetch_olivetti_faces()
>>> print faces.DESCR
```

The dataset contains 400 images of 40 different persons. The photos were taken with different light conditions and facial expressions (including open/closed eyes, smiling/not smiling, and with glasses/no glasses). For additional information about the dataset refer to <http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html>

Looking at the content of the `faces` object, we get the following properties: `images`, `data`, and `target`. `Images` contain the 400 images represented as 64 x 64 pixel matrices. `data` contains the same 400 images but as array of 4096 pixels. `target` is, as expected, an array with the target classes, ranging from 0 to 39.

```
>>> print faces.keys()
['images', 'data', 'target', 'DESCR']
>>> print faces.images.shape
(400, 64, 64)
>>> print faces.data.shape
(400, 4096)
>>> print faces.target.shape
(400,)
```

Normalizing the data is important as we saw in the previous chapter. It is also important for the application of SVM to obtain good results. In our particular case, we can verify by running the following snippet that our images already come as values in a very uniform range between 0 and 1 (pixel value):

```
>>> print np.max(faces.data)
1.0
>>> print np.min(faces.data)
0.0
>>> print np.mean(faces.data)
0.547046432495
```

Therefore, we do not have to normalize the data. Before learning, let's plot some faces. We will define the following helper function:

```
>>> def print_faces(images, target, top_n):
>>>     # set up the figure size in inches
>>>     fig = plt.figure(figsize=(12, 12))
>>>     fig.subplots_adjust(left=0, right=1,
bottom=0, top=1,
           hspace=0.05, wspace=0.05)
>>>     for i in range(top_n):
>>>         # plot the images in a matrix of
20x20
>>>         p = fig.add_subplot(20, 20, i + 1,
xticks=[],
           yticks[])
>>>         p.imshow(images[i],
cmap=plt.cm.bone)
>>>
>>>         # label the image with the target
value
>>>         p.text(0, 14, str(target[i]))
>>>         p.text(0, 60, str(i))
```

If we print the first 20 images, we can see faces from two

persons.

```
>>> print_faces(faces.images, faces.target, 20)
```



Training a Support Vector Machine

To use SVM in scikit-learn to solve our task, we will import the `SVC` class from the `sklearn.svm` module:

```
>>> from sklearn.svm import SVC
```

The **Support Vector Classifier (SVC)** will be used for classification. In the last section of this chapter, we will use SVM for regression tasks.

The SVC implementation has different important parameters; probably the most relevant is `kernel`, which defines the kernel function to be used in our classifier (think of the kernel functions as different similarity measures between instances). By default, the `SVC` class uses the `rbf` kernel, which allows us to model nonlinear problems. To start, we will use the simplest kernel, the linear one.

```
>>> svc_1 = SVC(kernel='linear')
```

Before continuing, we will split our dataset into training and testing datasets.

```
>>> from sklearn.cross_validation import  
train_test_split
```

```
>>> X_train, X_test, y_train, y_test =
train_test_split(
    faces.data, faces.target, test_size=0.25,
random_state=0)
```

And we will define a function to evaluate K-fold cross-validation.

```
>>> from sklearn.cross_validation import
cross_val_score, KFold
>>> from scipy.stats import sem
>>>
>>> def evaluate_cross_validation(clf, X, y, K):
>>>     # create a k-fold cross validation
iterator
>>>     cv = KFold(len(y), K, shuffle=True,
random_state=0)
>>>     # by default the score used is the one
returned by score
        method of the estimator (accuracy)
>>>     scores = cross_val_score(clf, X, y,
cv=cv)
>>>     print scores
>>>     print ("Mean score: {0:.3f} (+/-"
{1:.3f}).format(
        np.mean(scores), sem(scores))

>>> evaluate_cross_validation(svc_1, X_train,
y_train, 5)
[ 0.93333333  0.91666667  0.95          0.95
0.91666667]
Mean score: 0.933 (+/-0.007)
```

Cross-validation with five folds, obtains pretty good results (accuracy of 0.933). In a few steps we obtained a face classifier.

We will also define a function to perform training on the training set and evaluate the performance on the testing set.

```
>>> from sklearn import metrics
>>>
>>> def train_and_evaluate(clf, X_train, X_test,
y_train, y_test):
>>>
>>>     clf.fit(X_train, y_train)
>>>
>>>     print "Accuracy on training set:"
>>>     print clf.score(X_train, y_train)
>>>     print "Accuracy on testing set:"
>>>     print clf.score(X_test, y_test)
>>>
>>>     y_pred = clf.predict(X_test)
>>>
>>>     print "Classification Report:"
>>>     print
metrics.classification_report(y_test, y_pred)
>>>     print "Confusion Matrix:"
>>>     print metrics.confusion_matrix(y_test,
y_pred)
```

If we train and evaluate, the classifier performs the operation with almost no errors.

```
>>> train_and_evaluate(svc_1, X_train, X_test,  
y_train, y_test)  
Accuracy on training set:  
1.0  
Accuracy on testing set:  
0.99
```

Let's do a little more, why don't we try to classify the faces as people with and without glasses? Let's do that.

First thing to do is to define the range of the images that show faces wearing glasses. The following list shows the indexes of these images:

```
>>> # the index ranges of images of people with  
glasses  
>>> glasses = [  
    (10, 19), (30, 32), (37, 38), (50, 59), (63,  
64),  
    (69, 69), (120, 121), (124, 129), (130, 139),  
(160, 161),  
    (164, 169), (180, 182), (185, 185), (189,  
189), (190, 192),  
    (194, 194), (196, 199), (260, 269), (270,  
279), (300, 309),  
    (330, 339), (358, 359), (360, 369)  
]
```

You can check these values by using the `print_faces` function that was defined before to plot the 400 faces and looking at the indexes in the lower-left corners.

Then we'll define a function that from those segments returns a new target array that marks with 1 for the faces with glasses and 0 for the faces without glasses (our new target classes):

```
>>> def create_target(segments):
>>>     # create a new y array of target size
initialized with
    zeros
>>>     y = np.zeros(faces.target.shape[0])
>>>     # put 1 in the specified segments
>>>     for (start, end) in segments:
>>>         y[start:end + 1] = 1
>>>     return y
>>> target_glasses = create_target(glasses)
```

So we must perform the training/testing split again.

```
>>> X_train, X_test, y_train, y_test =
train_test_split(
    faces.data, target_glasses,
test_size=0.25, random_state=0)
```

Now let's create a new SVC classifier, and train it with the new target vector using the following command:

```
>>> svc_2 = SVC(kernel='linear')
```

If we check the performance with cross-validation by the following code:

```
>>> evaluate_cross_validation(svc_2, X_train,
```

```
y_train, 5)
[ 0.98333333  0.98333333  0.93333333  0.96666667
0.96666667]
Mean score: 0.967 (+/-0.009)
```

We obtain a mean accuracy of 0.967 with cross-validation if we evaluate on our testing set.

```
>>> train_and_evaluate(svc_2, x_train, x_test,
y_train, y_test)
Accuracy on training set:
1.0
Accuracy on testing set:
0.99
Classification Report:
              precision      recall   f1-score
support

          0          1.00      0.99      0.99
67
          1          0.97      1.00      0.99
33

avg / total      0.99      0.99      0.99
100

Confusion Matrix:
[[66  1]
 [ 0 33]]
```

Could it be possible that our classifier has learned to identify peoples' faces associated with glasses and without

glasses precisely? How can we be sure that this is not happening and that if we get new unseen faces, it will work as expected? Let's separate all the images of the same person, sometimes wearing glasses and sometimes not. We will also separate all the images of the same person, the ones with indexes from 30 to 39, train by using the remaining instances, and evaluate on our new 10 instances set. With this experiment we will try to discard the fact that it is remembering faces, not glassed-related features.

```
>>> X_test = faces.data[30:40]
>>> y_test = target_glasses[30:40]
>>> print y_test.shape[0]
10
>>> select = np.ones(target_glasses.shape[0])
>>> select[30:40] = 0
>>> X_train = faces.data[select == 1]
>>> y_train = target_glasses[select == 1]
>>> print y_train.shape[0]
390
>>> svc_3 = SVC(kernel='linear')
>>> train_and_evaluate(svc_3, X_train, X_test,
y_train, y_test)
Accuracy on training set:
1.0
Accuracy on testing set:
0.9
Classification Report:
precision      recall    f1-score
support
```

	0	0.83	1.00	0.91
5	1	1.00	0.80	0.89
5				
avg / total		0.92	0.90	0.90
10				

Confusion Matrix:

```
[ [5 0]
  [1 4]]
```

From the 10 images, only one error, still pretty good results, let's check out which one was incorrectly classified. First, we have to reshape the data from arrays to 64 x 64 matrices:

```
>>> y_pred = svc_3.predict(X_test)
>>> eval_faces = [np.reshape(a, (64, 64)) for a
in X_test]
```

Then plot with our `print_faces` function:

```
>>> print_faces(eval_faces, y_pred, 10)
```



The image number **8** in the preceding figure has glasses and was classified as no glasses. If we look at that instance, we can see that it is different from the rest of the images with glasses (the border of the glasses cannot be seen clearly and the person is shown with closed eyes), which could be the reason it has been misclassified.

With a few lines, we created a face classifier with a linear SVM model. Usually we would not get such good results in the first trial. In these cases, (besides looking at different features) we can start tweaking the hyperparameters of our algorithm. In the particular case of SVM, we can try with different kernel functions; if linear does not give good results, we can try with polynomial or RBF kernels. Also the `c` and the `gamma` parameters may affect the results. For a description of the arguments and its values, please refer to the scikit-learn documentation.

Text classification with Naïve Bayes

Naïve Bayes is a simple but powerful classifier based on a probabilistic model derived from the Bayes' theorem. Basically it determines the probability that an instance belongs to a class based on each of the feature value probabilities. The naïve term comes from the fact that it assumes that each feature is independent of the rest, that is, the value of a feature has no relation to the value of another feature.

Despite being very simple, it has been used in many domains with very good results. The independence assumption, although a naïve and strong simplification, is one of the features that make the model useful in practical applications. Training the model is reduced to the calculation of the involved conditional probabilities, which can be estimated by counting frequencies of correlations between feature values and class values.

One of the most successful applications of Naïve Bayes has been within the field of **Natural Language Processing (NLP)**. NLP is a field that has been much related to machine learning, since many of its problems

can be formulated as a classification task. Usually, NLP problems have important amounts of tagged data in the form of text documents. This data can be used as a training dataset for machine learning algorithms.

In this section, we will use Naïve Bayes for text classification; we will have a set of text documents with their corresponding categories, and we will train a Naïve Bayes algorithm to learn to predict the categories of new unseen instances. This simple task has many practical applications; probably the most known and widely used one is **spam filtering**. In this section we will try to classify newsgroup messages using a dataset that can be retrieved from within scikit-learn. This dataset consists of around 19,000 newsgroup messages from 20 different topics ranging from politics and religion to sports and science.

As usual, we first start by importing our `pylab` environment:

```
>>> %pylab inline
```

Our dataset can be obtained by importing the `fetch_20newgroups` function from the `sklearn.datasets` module. We have to specify if we want to import a part or all of the set of instances (we will import all of them).

```
>>> from sklearn.datasets import  
fetch_20newsgroups  
>>> news = fetch_20newsgroups(subset='all')
```

If we look at the properties of the dataset, we will find that we have the usual ones: DESCR, data, target, and target_names. The difference now is that data holds a list of text contents, instead of a numpy matrix:

```
>>> print type(news.data), type(news.target),  
type(news.target_names)  
<type 'list'> <type 'numpy.ndarray'> <type  
'list'>  
>>> print news.target_names  
['alt.atheism', 'comp.graphics', 'comp.os.ms-  
windows.misc', 'comp.sys.ibm.pc.hardware',  
'comp.sys.mac.hardware', 'comp.windows.x',  
'misc.forsale', 'rec.autos', 'rec.motorcycles',  
'rec.sport.baseball', 'rec.sport.hockey',  
'sci.crypt', 'sci.electronics', 'sci.med',  
'sci.space', 'soc.religion.christian',  
'talk.politics.guns', 'talk.politics.mideast',  
'talk.politics.misc', 'talk.religion.misc']  
>>> print len(news.data)  
18846  
>>> print len(news.target)  
18846
```

If you look at, say, the first instance, you will see the content of a newsgroup message, and you can get its corresponding category:

```
>>> print news.data[0]
>>> print news.target[0],
news.target_names(news.target[0])
```

Preprocessing the data

Our machine learning algorithms can work only on numeric data, so our next step will be to convert our text-based dataset to a numeric dataset. Currently we only have one feature, the text content of the message; we need some function that transforms a text into a meaningful set of numeric features. Intuitively one could try to look at which are the words (or more precisely, tokens, including numbers or punctuation signs) that are used in each of the text categories, and try to characterize each category with the frequency distribution of each of those words. The `sklearn.feature_extraction.text` module has some useful utilities to build numeric feature vectors from text documents.

Before starting the transformation, we will have to partition our data into training and testing set. The loaded data is already in a random order, so we only have to split the data into, for example, 75 percent for training and the rest 25 percent for testing:

```
>>> SPLIT_PERC = 0.75
>>> split_size = int(len(news.data)*SPLIT_PERC)
>>> X_train = news.data[:split_size]
>>> X_test = news.data[split_size:]
>>> y_train = news.target[:split_size]
>>> y_test = news.target[split_size:]
```

If you look inside the `sklearn.feature_extraction.text` module, you will find three different classes that can transform text into numeric features: `CountVectorizer`, `HashingVectorizer`, and `TfidfVectorizer`. The difference between them resides in the calculations they perform to obtain the numeric features. `CountVectorizer` basically creates a dictionary of words from the text corpus. Then, each instance is converted to a vector of numeric features where each element will be the count of the number of times a particular word appears in the document.

`HashingVectorizer`, instead of constricting and maintaining the dictionary in memory, implements a hashing function that maps tokens into feature indexes, and then computes the count as in `CountVectorizer`.

`TfidfVectorizer` works like the `CountVectorizer`, but with a more advanced calculation called **Term Frequency Inverse Document Frequency (TF-IDF)**. This is a statistic for measuring the importance of a word in a document or corpus. Intuitively, it looks for words that are more frequent in the current document, compared with their frequency in the whole corpus of documents. You can see this as a way to normalize the results and avoid words that are too frequent, and thus not useful to characterize the instances.

Training a Naïve Bayes classifier

We will create a Naïve Bayes classifier that is composed of a feature vectorizer and the actual Bayes classifier. We will use the `MultinomialNB` class from the `sklearn.naive_bayes` module. In order to compose the classifier with the vectorizer, as we saw in [Chapter 1](#), *Machine Learning – A Gentle Introduction*, scikit-learn has a very useful class called `Pipeline` (available in the `sklearn.pipeline` module) that eases the construction of a compound classifier, which consists of several vectorizers and classifiers.

We will create three different classifiers by combining `MultinomialNB` with the three different text vectorizers just mentioned, and compare which one performs better using the default parameters:

```
>>> from sklearn.naive_bayes import  
MultinomialNB  
>>> from sklearn.pipeline import Pipeline  
>>> from sklearn.feature_extraction.text import  
TfidfVectorizer,  
  
HashingVectorizer, CountVectorizer  
>>> clf_1 = Pipeline([  
>>>     ('vect', CountVectorizer()),  
>>>     ('clf', MultinomialNB()),  
>>> ])
```

```
>>> clf_2 = Pipeline([
>>>     ('vect',
HashingVectorizer(non_negative=True)),
>>>     ('clf', MultinomialNB()),
>>> ])
>>> clf_3 = Pipeline([
>>>     ('vect', TfidfVectorizer()),
>>>     ('clf', MultinomialNB()),
>>> ])
```

We will define a function that takes a classifier and performs the K-fold cross-validation over the specified x and y values:

```
>>> from sklearn.cross_validation import
cross_val_score, KFold
>>> from scipy.stats import sem
>>>
>>> def evaluate_cross_validation(clf, X, y, K):
>>>     # create a k-fold cross validation
iterator of k=5 folds
>>>     cv = KFold(len(y), K, shuffle=True,
random_state=0)
>>>     # by default the score used is the one
returned by score >>>     method of the
estimator (accuracy)
>>>     scores = cross_val_score(clf, X, y,
cv=cv)
>>>     print scores
>>>     print ("Mean score: {:.3f} (+/-"
{1:.3f})".format(
>>>             np.mean(scores), sem(scores))
```

Then we will perform a five-fold cross-validation by using each one of the classifiers.

```
>>> clfs = [clf_1, clf_2, clf_3]
>>> for clf in clfs:
>>>     evaluate_cross_validation(clf,
news.data, news.target, 5)
```

These calculations may take some time; the results are as follows:

```
[ 0.86813478  0.86415495  0.86893075  0.85831786
0.8729443 ]
Mean score: 0.866 (+/-0.002)
[ 0.76359777  0.77182276  0.77765986  0.76147519
0.78222812]
Mean score: 0.771 (+/-0.004)
[ 0.86282834  0.85195012  0.86282834  0.85619528
0.87612732]
Mean score: 0.862 (+/-0.004)
```

As you can see CountVectorizer and TfidfVectorizer had similar performances, and much better than HashingVectorizer.

Let's continue with TfidfVectorizer; we could try to improve the results by trying to parse the text documents into tokens with a different regular expression.

```
>>> clf_4 = Pipeline([
>>>     ('vect', TfidfVectorizer(
```

```
>>>         token_pattern=ur"\b[a-zA-Z0-9_\\-\\.]+  
[a-zA-Z0-9_-  
>>>             \.]+\b",  
>>>     ),  
>>>     ('clf', MultinomialNB()),  
>>> ])
```

The default regular expression: `ur"\b\w\w+\b"` considers alphanumeric characters and the underscore. Perhaps also considering the slash and the dot could improve the tokenization, and begin considering tokens as *Wi-Fi* and [site.com](#). The new regular expression could be: `ur"\b[a-zA-Z0-9_\\-\\.]+[a-zA-Z][a-zA-Z0-9_\\-\\.]+\\b"`. If you have queries about how to define regular expressions, please refer to the Python `re` module documentation. Let's try our new classifier:

```
>>> evaluate_cross_validation(clf_4, news.data,  
news.target, 5)  
[ 0.87078801  0.86309366  0.87689042  0.86574688  
0.8795756 ]  
Mean score: 0.871 (+/-0.003)
```

We have a slight improvement from 0.86 to 0.87.

Another parameter that we can use is `stop_words`: this argument allows us to pass a list of words we do not want to take into account, such as too frequent words, or words we do not a priori expect to provide information about the

particular topic.

We will define a function to load the stop words from a text file as follows:

```
>>> def get_stop_words():
>>>     result = set()
>>>     for line in open('stopwords_en.txt',
'r').readlines():
>>>         result.add(line.strip())
>>>     return result
```

And create a new classifier with this new parameter as follows:

```
>>> clf_5 = Pipeline([
>>>     ('vect', TfidfVectorizer(
>>>             stop_words=
get_stop_words(),
>>>             token_pattern=ur"\b[a-zA-Z0-
9_\.]+\b",
>>>             )),
>>>     ('clf', MultinomialNB()),
>>> ])
>>> evaluate_cross_validation(clf_5, news.data,
news.target, 5)
[ 0.88989122  0.8837888   0.89042186  0.88325816
0.89655172]
Mean score: 0.889 (+/-0.002)
```

The preceding code shows another improvement from 0.87 to 0.89.

Let's keep this vectorizer and start looking at the `MultinomialNB` parameters. This classifier has few parameters to tweak; the most important is the `alpha` parameter, which is a smoothing parameter. Let's set it to a lower value; instead of setting alpha to 1.0 (the default value), we will set it to 0.01:

```
>>> clf_7 = Pipeline([
>>>     ('vect', TfidfVectorizer(
>>>         stop_words=get_stop_words()
>>>         token_pattern=ur"\b[a-z0-
9_\.]+\b",
>>>         )),
>>>     ('clf', MultinomialNB(alpha=0.01)),
>>> ])
```



```
>>> evaluate_cross_validation(clf_7, news.data,
news.target, 5)
[ 0.92305651  0.91377023  0.92066861  0.91907668
0.92281167]
Mean score: 0.920 (+/-0.002)
```

The results had an important boost from 0.89 to 0.92, pretty good. At this point, we could continue doing trials by using different values of alpha or doing new modifications of the vectorizer. In [Chapter 4, Advanced](#)

Features, we will show you practical utilities to try many different configurations and keep the best one. But for now, let's look a little more at our Naïve Bayes model.

Evaluating the performance

If we decide that we have made enough improvements in our model, we are ready to evaluate its performance on the testing set.

We will define a helper function that will train the model in the entire training set and evaluate the accuracy in the training and in the testing sets. It will also print a classification report (precision and recall on every class) and the corresponding confusion matrix:

```
>>> from sklearn import metrics
>>>
>>> def train_and_evaluate(clf, X_train, X_test,
y_train, y_test):
>>>
>>>     clf.fit(X_train, y_train)
>>>
>>>     print "Accuracy on training set:"
>>>     print clf.score(X_train, y_train)
>>>     print "Accuracy on testing set:"
>>>     print clf.score(X_test, y_test)
>>>     y_pred = clf.predict(X_test)
>>>
>>>     print "Classification Report:"
>>>     print
metrics.classification_report(y_test, y_pred)
>>>     print "Confusion Matrix:"
>>>     print metrics.confusion_matrix(y_test,
y_pred)
```

We will evaluate our best classifier.

```
>>> train_and_evaluate(clf_7, X_train, X_test,  
y_train, y_test)  
Accuracy on training set:  
0.99398613273  
Accuracy on testing set:  
0.913837011885
```

As we can see, we obtained very good results, and as we would expect, the accuracy in the training set is quite better than in the testing set. We may expect, in new unseen instances, an accuracy of around 0.91.

If we look inside the vectorizer, we can see which tokens have been used to create our dictionary:

```
>>> print  
len(clf_7.named_steps['vect'].get_feature_names()  
)  
61236
```

This shows that the dictionary is composed of 61236 tokens. Let's print the feature names.

```
>>>  
clf_7.named_steps['vect'].get_feature_names()
```

The following table presents an extract of the results:



Extract of features obtained by vectorizer

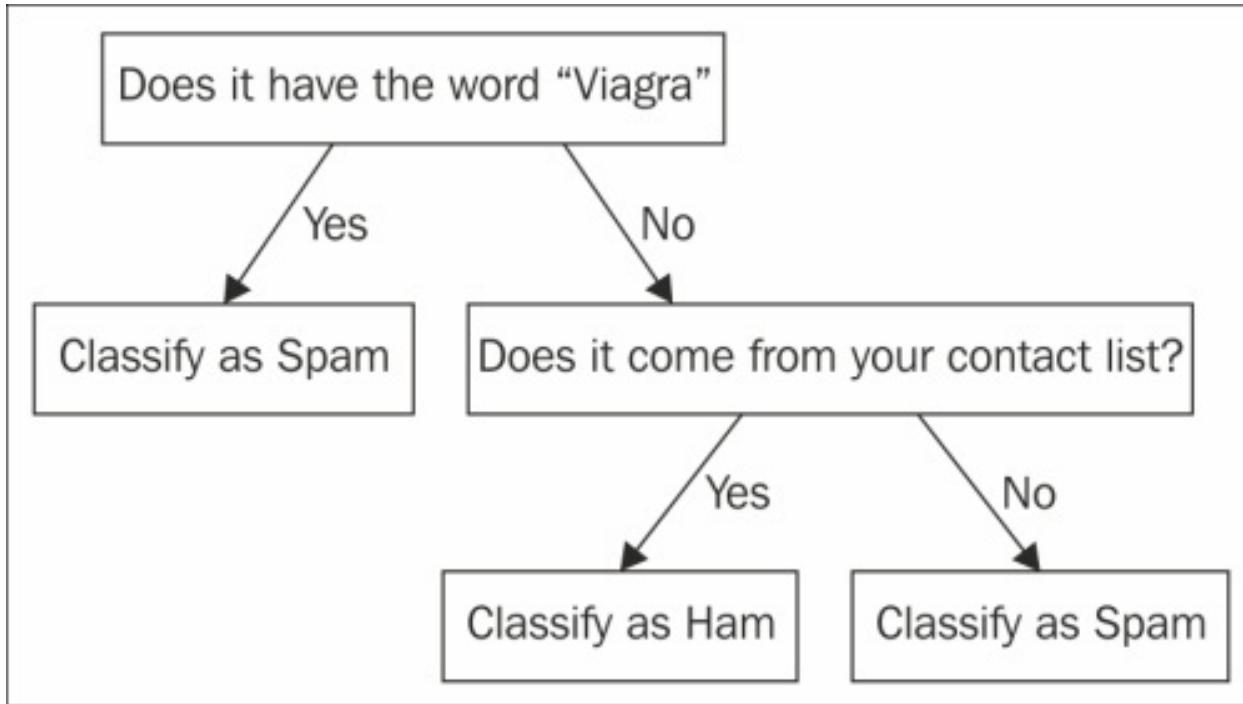
u'''sanctuaries''', u'''sanctuary''', u'''sanctum''', u'''sand''', u'''sandals''', u'''sandbags''', u'''sandberg''', u'''sandblasting''', u'''anders'''	u'''sanderson''', u'''sandia''', u'''sandiego.ncr.com''', u'''sanding''', u'''sandlak''', u'''sandman.caltech.edu''', u'''sandman.ece.clarkson.edu''', u'''sandra''', u'''sandro''', u'''sands'''
---	--

You can see that some words are semantically very similar, for example, sand and sands, sanctuaries and sanctuary. Perhaps if the plurals and the singulars are counted to the same bucket, we would better represent the documents. This is a very common task, which could be solved using stemming, a technique that relates two words having the same lexical root.

Explaining Titanic hypothesis with decision trees

A common argument against linear classifiers and against statistical learning methods is that it is difficult to explain how the built model decides its predictions for the target classes. If you have a highly dimensional SVM, it is impossible for a human being to even imagine how the hyperplane built looks like. A Naïve Bayes classifier will tell you something like: "this class is the most probable, assuming it comes from a similar distribution as the training data, and making a few more assumptions" something not very useful, for example, we want to know why this or that mail should be considered as spam.

decision trees are very simple yet powerful supervised learning methods, which constructs a decision tree model, which will be used to make predictions. The following figure shows a very simple decision tree to decide if an e-mail should be considered spam:



It first asks if the e-mail contains the word **Viagra**; if the answer is yes, it classifies it as spam; if the answer is no, it further asks if it comes from somebody in your contacts list; this time, if the answer is yes, it classifies the e-mail as Ham; if the answer is no, it classify it as spam. The main advantage of this model is that a human being can easily understand and reproduce the sequence of decisions (especially if the number of attributes is small) taken to predict the target class of a new instance. This is very important for tasks such as medical diagnosis or credit approval, where we want to show a reason for the decision, rather than just saying this is what the training

data suggests (which is, by definition, what every supervised learning method does). In this section, we will show you through a working example what decision trees look like, how they are built, and how they are used for prediction.

The problem we would like to solve is to determine if a Titanic's passenger would have survived, given her age, passenger class, and sex. We will work with the Titanic dataset that can be downloaded from

<http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic.csv>

Like every other example in this chapter, we start with a dataset that includes the list of Titanic's passengers and a feature indicating whether they survived or not. Each instance in the dataset has the following form:

```
"1","1st",1,"Allen, Miss Elisabeth Walton",29.0000,"Southampton","St Louis, MO","B-5","24160 L221","2","female"
```

The list of attributes is: Ordinal, Class, Survived (0=no, 1=yes), Name, Age, Port of Embarkation, Home/Destination, Room, Ticket, Boat, and Sex. We will start by loading the dataset into a numpy array.

```
>>> import csv  
>>> import numpy as np  
>>> with open('data/titanic.csv', 'rb') as csvfile:
```

```
>>> titanic_reader = csv.reader(csvfile,
delimiter=',',
quotechar='''')
>>>
>>> # Header contains feature names
>>> row = titanic_reader.next()
>>> feature_names = np.array(row)
>>>
>>> # Load dataset, and target classes
>>> titanic_X, titanic_y = [], []
>>> for row in titanic_reader:
>>>     titanic_X.append(row)
>>>     titanic_y.append(row[2]) # The
target value is
        "survived"
>>>
>>> titanic_X = np.array(titanic_X)
>>> titanic_y = np.array(titanic_y)
```

The code shown uses the Python `csv` module to load the data.

```
>>> print feature_names
['row.names' 'pclass' 'survived' 'name' 'age'
'embarked' 'home.dest' 'room' 'ticket' 'boat'
'sex']

>>> print titanic_X[0], titanic_y[0]
['1' '1st' '1' 'Allen, Miss Elisabeth Walton'
'29.0000' 'Southampton' 'St Louis, MO' 'B-5'
'24160 L221' '2' 'female'] 1
```

Preprocessing the data

The first step we must take is to select the attributes we will use for learning:

```
>>> # we keep class, age and sex  
>>> titanic_X = titanic_X[:, [1, 4, 10]]  
>>> feature_names = feature_names[[1, 4, 10]]
```

We have selected feature numbers 1, 4, and 10 that is class, age, and sex, based on the assumption that the remaining attributes have no effect on the passenger's survival. Feature selection is an extremely important step while creating a machine learning solution. If the algorithm does not have good features as input, it will not have good enough material to learn from, results won't be good, no matter even if we have the best machine learning algorithm ever designed.

Sometimes the feature selection will be made manually, based on our knowledge of the problem's domain and the machine learning method we are planning to use. Sometimes feature selection may be done by using automatic tools to evaluate and select the most promising ones. In [Chapter 4, Advanced Features](#), we will talk a bit about these techniques, but for now, we will manually select our attributes. Very specific attributes (such as Name

in our case) could result in overfitting (consider a tree that just asks if the name is X, she survived); attributes where there is a small number of instances with each value, present a similar problem (they might not be useful for generalization). We will use class, age, and sex because a priori, we expect them to have influenced the passenger's survival.

Now, our learning data looks like:

```
>>> print feature_names  
['pclass' 'age' 'sex']  
  
>>> print titanic_X[12],titanic_y[12]  
['1st' 'NA' 'female'] 1
```

We have shown instance number 12 because it poses a problem to solve; one of its features (the age) is not available. We have **missing values**, a usual problem with datasets. In this case, we decided to substitute missing values with the mean age in the training data. We could have taken a different approach, for example, using the most common value in the training data, or the median value. When we substitute missing values, we have to understand that we are modifying the original problem, so we have to be very careful with what we are doing. This is a general rule in machine learning; when we change data, we should have a clear idea of what we are

changing, to avoid skewing the final results.

```
>>> # We have missing values for age
>>> # Assign the mean value
>>> ages = titanic_X[:, 1]
>>> mean_age = np.mean(titanic_X[ages != 'NA',
1].astype(np.float))
>>> titanic_X[titanic_X[:, 1] == 'NA', 1] =
mean_age
```

The implementation of decision trees in scikit-learn expects as input a list of real-valued features, and the decision rules of the model would be of the form:

Feature \leq value

For example, $\text{age} \leq 20.0$. Our attributes (except for age) are categorical; that is, they correspond to a value taken from a discrete set such as male and female. So, we have to convert categorical data into real values. Let's start with the sex feature. The preprocessing module of scikit-learn includes a `LabelEncoder` class, whose `fit` method allows conversion of a categorical set into a $0 \dots K-1$ integer, where K is the number of different classes in the set (in the case of sex, just 0 or 1):

```
>>> # Encode sex
>>> from sklearn.preprocessing import
LabelEncoder
>>> enc = LabelEncoder()
```

```
>>> label_encoder = enc.fit(titanic_X[:, 2])
>>> print "Categorical classes:",
label_encoder.classes_
Categorical classes: ['female' 'male']

>>> integer_classes =
label_encoder.transform(label_encoder.classes_)
>>> print "Integer classes:", integer_classes
Integer classes: [0 1]

>>> t = label_encoder.transform(titanic_X[:, 2])
>>> titanic_X[:, 2] = t
```

The last two sentences transform the values of the sex attribute into 0-1 values, and modify the training set.

```
print feature_names
['pclass' 'age' 'sex']

print titanic_X[12], titanic_y[12]
['1st' '31.1941810427' '0'] 1
```

We still have a categorical attribute: `class`. We could use the same approach and convert its three classes into 0, 1, and 2. This transformation implicitly introduces an ordering between classes, something that is not an issue in our problem. However, we will try a more general approach that does not assume an ordering, and it is widely used to convert categorical classes into real-valued attributes. We will introduce an additional encoder and

convert the class attributes into three new binary features, each of them indicating if the instance belongs to a feature value (1) or (0). This is called **one hot encoding**, and it is a very common way of managing categorical attributes for real-based methods:

```
>>> from sklearn.preprocessing import  
OneHotEncoder  
>>>  
>>> enc = LabelEncoder()  
>>> label_encoder = enc.fit(titanic_X[:, 0])  
>>> print "Categorical classes:",  
label_encoder.classes_  
Categorical classes: ['1st' '2nd' '3rd']  
  
>>> integer_classes =  
  
label_encoder.transform(label_encoder.classes_).  
reshape(3, 1)  
>>> print "Integer classes:", integer_classes  
Integer classes: [[0] [1] [2]]  
  
>>> enc = OneHotEncoder()  
>>> one_hot_encoder = enc.fit(integer_classes)  
>>> # First, convert classes to 0-(N-1) integers  
using  
    label_encoder  
>>> num_of_rows = titanic_X.shape[0]  
>>> t = label_encoder.transform(titanic_X[:,  
    0]).reshape(num_of_rows, 1)  
>>> # Second, create a sparse matrix with three  
columns, each one
```

```
    indicating if the instance belongs to the
    class
>>> new_features = one_hot_encoder.transform(t)
>>> # Add the new features to titanic_X
>>> titanic_X = np.concatenate([titanic_X,
    new_features.toarray()], axis = 1)
>>> #Eliminate converted columns
>>> titanic_X = np.delete(titanic_X, [0], 1)
>>> # Update feature names
>>> feature_names = ['age', 'sex',
    'first_class', 'second_class',
    'third_class']
>>> # Convert to numerical values
>>> titanic_X = titanic_X.astype(float)
>>> titanic_y = titanic_y.astype(float)
```

The preceding code first converts the classes into integers and then uses the `OneHotEncoder` class to create the three new attributes that are added to the array of features. It finally eliminates from training data the original class feature.

```
>>> print feature_names
['age', 'sex', 'first_class', 'second_class',
'third_class']

>>> print titanic_X[0], titanic_y[0]
[29.  0.  1.  0.  0.] 1.0
```

We have now a suitable learning set for scikit-learn to learn a decision tree. Also, standardization is not an issue

for decision trees because the relative magnitude of features does not affect the classifier performance.

The preprocessing step is usually underestimated in machine learning methods, but as we can see even in this very simple example, it can take some time to make data look as our methods expect. It is also very important in the overall machine learning process; if we fail in this step (for example, incorrectly encoding attributes, or selecting the wrong features), the following steps will fail, no matter how good the method we use for learning.

Training a decision tree classifier

Now to the interesting part; let's build a decision tree from our training data. As usual, we will first separate training and testing data.

```
>>> from sklearn.cross_validation import  
train_test_split  
>>> X_train, X_test, y_train, y_test =  
train_test_split(titanic_X, >>> titanic_y,  
test_size=0.25, random_state=33)
```

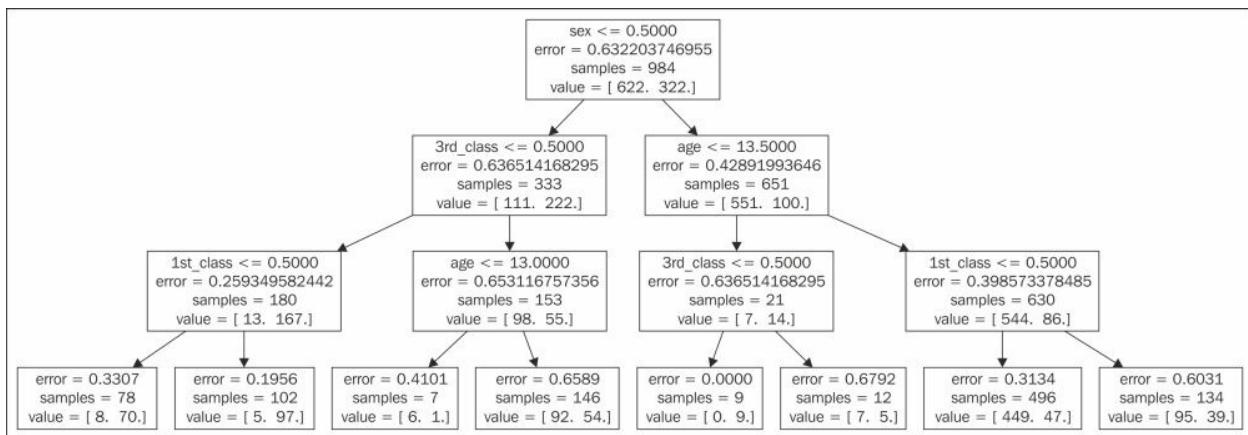
Now, we can create a new `DecisionTreeClassifier` and use the `fit` method of the classifier to do the learning job.

```
>>> from sklearn import tree  
>>> clf =  
tree.DecisionTreeClassifier(criterion='entropy',  
  
    max_depth=3, min_samples_leaf=5)  
>>> clf = clf.fit(X_train, y_train)
```

`DecisionTreeClassifier` accepts (as most learning methods) several hyperparameters that control its behavior. In this case, we used the **Information Gain (IG)** criterion for splitting learning data, told the method to build a tree of at most three levels, and to accept a node as a leaf if it includes at least five training instances. To explain this and show how decision trees work, let's

visualize the model built. The following code assumes you are using IPython and that your Python distribution includes the `pydot` module. Also, it allows generation of **Graphviz** code from the tree and assumes that Graphviz itself is installed. For more information about Graphviz, please refer to <http://www.graphviz.org/>.

```
>>> import pydot, StringIO
>>> dot_data = StringIO.StringIO()
>>> tree.export_graphviz(clf, out_file=dot_data,
    feature_names=
['age','sex','1st_class','2nd_class',
 '3rd_class'])
>>> graph =
pydot.graph_from_dot_data(dot_data.getvalue())
>>> graph.write_png('titanic.png')
>>> from IPython.core.display import Image
>>> Image(filename='titanic.png')
```



The decision tree we have built represents a series of

decisions based on the training data. To classify an instance, we should answer the question at each node. For example, at our root node, the question is: Is $\text{sex} \leq 0.5$? (are we talking about a woman?). If the answer is yes, you go to the left child node in the tree; otherwise you go to the right child node. You keep answering questions (was she in the third class?, was she in the first class?, and was she below 13 years old?), until you reach a leaf. When you are there, the prediction corresponds to the target class that has most instances (that is if the answers are given to the previous questions). In our case, if she was a woman from second class, the answer would be 1 (that is she survived), and so on.

You might be asking how our method decides which questions should be asked in each step. The answer is **Information Gain (IG)** (or the Gini index, which is a similar measure of disorder used by scikit-learn). IG measures how much entropy we lose if we answer the question, or alternatively, how much surer we are after answering it. **Entropy** is a measure of disorder in a set, if we have zero entropy, it means all values are the same (in our case, all instances of the target classes are the same), while it reaches its maximum when there is an equal number of instances of each class (in our case, when half of the instances correspond to survivors and the other half to non survivors). At each node, we have a certain number

of instances (starting from the whole dataset), and we measure its entropy. Our method will select the questions that yield more homogeneous partitions (with the lowest entropy), when we consider only those instances for which the answer for the question is yes or no, that is, when the entropy after answering the question decreases.

Interpreting the decision tree

As you can see in the tree, at the beginning of the decision tree growing process, you have the 984 instances in the training set, 662 of them corresponding to class 0 (fatalities), and 322 of them to class 1 (survivors). The measured entropy for this initial group is about 0.632. From the possible list of questions we can ask, the one that produces the greatest information gain is: Was she a woman? (remember that the female category was encoded as 0). If the answer is yes, entropy is almost the same, but if the answer is no, it is greatly reduced (the proportion of men who died was much greater than the general proportion of casualties). In this sense, the woman question seems to be the best to ask. After that, the process continues, working in each node only with the instances that have feature values that correspond to the questions in the path to the node.

If you look at the tree, in each node we have: the question, the initial Shannon entropy, the number of instances we are considering, and their distribution with respect to the target class. In each step, the number of instances gets reduced to those that answer yes (the left branch) and no (the right branch) to the question posed by that node. The process continues until a certain stopping criterion is met

(in our case, until we have a fourth-level node, or the number of considered samples is lower than five).

At prediction time, we take an instance and start traversing the tree, answering the questions based on the instance features, until we reach a leaf. At this point, we look at how many instances of each class we had in the training set, and select the class to which most instances belonged.

For example, consider the question of determining if a 10-year-old girl, from first class would have survived. The answer to the first question (was she female?) is yes, so we take the left branch of the tree. In the two following questions the answers are no (was she from third class?) and yes (was she from first class?), so we take the left and right branch respectively. At this time, we have reached a leaf. In the training set, we had 102 people with these attributes, 97 of them survivors. So, our answer would be survived.

In general, we found reasonable results: the group with more casualties (449 from 496) corresponded to adult men from second or third class, as you can check in the tree. Most girls from first class, on the other side, survived. Let's measure the accuracy of our method in the training set (we will first define a helper function to

measure the performance of a classifier):

```
>>> from sklearn import metrics
>>> def measure_performance(X,y,clf,
show_accuracy=True,
    show_classification_report=True,
show_confussion_matrix=True):
>>>     y_pred=clf.predict(X)
>>>     if show_accuracy:
>>>         print "Accuracy:{0:.3f}".format(
>>>             metrics.accuracy_score(y,
y_pred)
>>>         ),"\n"
>>>
>>>     if show_classification_report:
>>>         print "Classification report"
>>>         print
metrics.classification_report(y,y_pred),"\n"
>>>
>>>     if show_confussion_matrix:
>>>         print "Confussion matrix"
>>>         print
metrics.confusion_matrix(y,y_pred),"\n"

>>> measure_performance(X_train,y_train,clf,
    show_classification=False,
show_confussion_matrix=False))
Accuracy:0.838
```

Our tree has an accuracy of 0.838 on the training set. But remember that this is not a good indicator. This is especially true for decision trees as this method is highly

susceptible to overfitting. Since we did not separate an evaluation set, we should apply cross-validation. For this example, we will use an extreme case of cross-validation, named **leave-one-out cross-validation**. For each instance in the training sample, we train on the rest of the sample, and evaluate the model built on the only instance left out. After performing as many classifications as training instances, we calculate the accuracy simply as the proportion of times our method correctly predicted the class of the left-out instance, and found it is a little lower (as we expected) than the resubstitution accuracy on the training set.

```
>>> from sklearn.cross_validation import  
cross_val_score, LeaveOneOut  
>>> from scipy.stats import sem  
>>>  
>>> def loo_cv(X_train, y_train, clf):  
>>>     # Perform Leave-One-Out cross validation  
>>>     # We are preforming 1313  
classifications!  
>>>     loo = LeaveOneOut(X_train[:,].shape[0])  
>>>     scores = np.zeros(X_train[:,].shape[0])  
>>>     for train_index, test_index in loo:  
>>>         X_train_cv, X_test_cv =  
X_train[train_index],  
                X_train[test_index]  
>>>         y_train_cv, y_test_cv =  
y_train[train_index],  
                y_train[test_index]  
>>>         clf = clf.fit(X_train_cv, y_train_cv)
```

```
>>> y_pred = clf.predict(X_test_cv)
>>> scores[test_index] =
metrics.accuracy_score(
    y_test_cv.astype(int), y_pred.astype(int))
>>> print ("Mean score: {0:.3f} (+/-"
{1:.3f})).format(np.mean(scores), sem(scores))

>>> loo_cv(X_train, y_train,clf)
Mean score: 0.837 (+/-0.012)
```

The main advantage of leave-one-out cross-validation is that it allows almost as much data for training as we have available, so it is particularly well suited for those cases where data is scarce. Its main problem is that training a different classifier for each instance could be very costly in terms of the computation time.

A big question remains here: how we selected the hyperparameters for our method instantiation? This problem is a general one, it is called model selection, and we will address it in more detail in [Chapter 4, Advanced Features](#).

Random Forests – randomizing decisions

A common criticism to decision trees is that once the training set is divided after answering a question, it is not possible to reconsider this decision. For example, if we divide men and women, every subsequent question would be only about men or women, and the method could not consider another type of question (say, age less than a year, irrespective of the gender). **Random Forests** try to introduce some level of randomization in each step, proposing alternative trees and combining them to get the final prediction. These types of algorithms that consider several classifiers answering the same question are called **ensemble methods**. In the Titanic task, it is probably hard to see this problem because we have very few features, but consider the case when the number of features is in the order of thousands.

Random Forests propose to build a decision tree based on a subset of the training instances (selected randomly, with replacement), but using a small random number of features at each set from the feature set. This tree growing process is repeated several times, producing a set of classifiers. At prediction time, each grown tree, given an instance, predicts its target class exactly as decision trees

do. The class that most of the trees vote (that is the class most predicted by the trees) is the one suggested by the ensemble classifier.

In scikit-learn, using Random Forests is as simple as importing `RandomForestClassifier` from the `sklearn.ensemble` module, and fitting the training data as follows:

```
>>> from sklearn.ensemble import  
RandomForestClassifier  
>>> clf =  
RandomForestClassifier(n_estimators=10,  
random_state=33)  
>>> clf = clf.fit(X_train, y_train)  
>>> loo_cv(X_train, y_train, clf)  
Mean score: 0.817 (+/-0.012)
```

We find that results are actually worse for Random Forests. It seems that introducing randomization was, after all, not a good idea because the number of features was too small. However, for bigger datasets, with a bigger number of features, Random Forests is a very fast, simple, and popular method to improve accuracy, retaining the virtues of decision trees. Actually, in the next section, we will use them for regression.

Evaluating the performance

The final step in every supervised learning task should be to evaluate our best classifier on the previously unseen data, to get an idea of its prediction performance.

Remember, this step should not be used to select among competing methods or parameters. That would be cheating (because again, we risk overfitting the new data). So, in our case, let's measure the performance of decision trees on the testing data.

```
>>> clf_dt =  
tree.DecisionTreeClassifier(criterion='entropy',  
max_depth=3, min_samples_leaf=5)  
>>> clf_dt.fit(X_train, y_train)  
>>> measure_performance(X_test, y_test, clf_dt)  
Accuracy:0.793  
Classification report  
          precision      recall   f1-score  
support  


|             | precision | recall | f1-score |
|-------------|-----------|--------|----------|
| 0           | 0.77      | 0.96   | 0.85     |
| 202         |           |        |          |
| 1           | 0.88      | 0.54   | 0.67     |
| 127         |           |        |          |
| avg / total | 0.81      | 0.79   | 0.78     |
| 329         |           |        |          |



Confusion matrix



```
[[193 9]
 [59 68]]
```


```

From the classification results and the confusion matrix, it seems that our method tends to predict too much that the person did not survive.

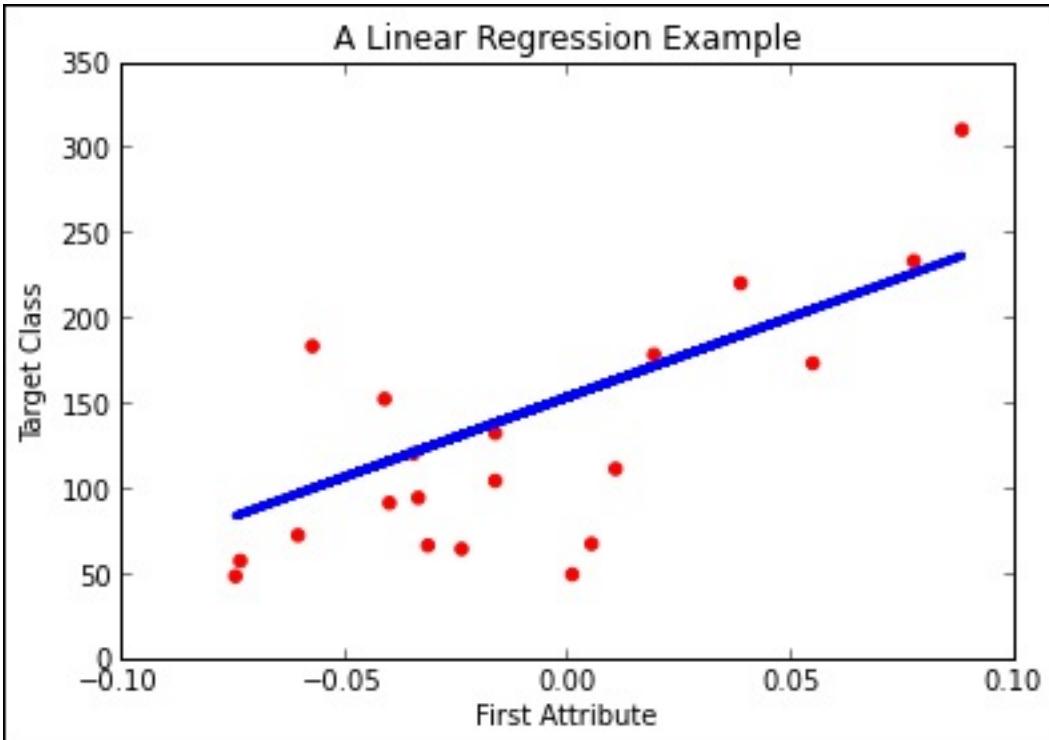
Predicting house prices with regression

In every example we have seen so far, we have faced what in [Chapter 1](#), *Machine Learning – A Gentle Introduction*, we called classification problems: the output we aimed at predicting belonged to a discrete set. But often, we would want to predict a value extracted from the real line. The learning schema is still the same: fit a model to the training data, and evaluate on new data to get the target class whose value is a real number. Our classifier, instead of selecting a class from a list, should act as a real-valued function, which for each of the (possibly infinite) combination of learning features returns a real number. We could consider regression as classification with an infinite number of target classes.

Many problems can be modeled both as classification and regression tasks, depending on the class we selected as the target. For example, predicting blood sugar level is a regression task, while predicting if somebody has diabetes or not is a classification task.

In the example of the first figure, we have used a line to fit the learning data (composed of a sole attribute and a

target value), that is, we have performed linear regression. If we want to predict the value of a new instance, we get their real-valued attribute and obtain the predicted value by projecting the inferred line into the second axis.



In this section, we will compare several regression methods by using the same dataset. We will try to predict the price of a house as a function of its attributes. As the dataset, we will use the Boston house-prices dataset, which includes 506 instances, representing houses in the suburbs of Boston by 14 features, one of them (the median value of owner-occupied homes) being the target

class (for a detailed reference, see <http://archive.ics.uci.edu/ml/datasets/Housing>). Each attribute in this dataset is real-valued.

The dataset is included in the standard scikit-learn distribution, so let's start by loading it:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
>>> print boston.data.shape
(506, 13)
>>> print boston.feature_names
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE'
 'DIS' 'RAD' 'TAX' 'PTRATIO' 'B' 'LSTAT' 'MEDV']
>>> print np.max(boston.target),
np.min(boston.target),
    np.mean(boston.target)
50.0 5.0 22.5328063241
```

You should try printing `boston.DESCR` to get a feel of what each feature means. This is a very healthy habit: machine learning is not just number crunching, understanding the problem we are facing is crucial, especially to select the best learning model to use.

As usual, we start slicing our learning set into training and testing datasets, and normalizing the data:

```
>>> from sklearn.cross_validation import
```

```
train_test_split
>>> X_train, X_test, y_train, y_test =
    train_test_split(boston.data, boston.target,
test_size=0.25,
    random_state=33)
>>> from sklearn.preprocessing import
StandardScaler
>>> scalerX = StandardScaler().fit(X_train)
>>> scalery = StandardScaler().fit(y_train)
>>> X_train = scalerX.transform(X_train)
>>> y_train = scalery.transform(y_train)
>>> X_test = scalerX.transform(X_test)
>>> y_test = scalery.transform(y_test)
```

Before looking at our best classifier, let's define how we will compare our results. Since we want to preserve our testing set for evaluating the performance of the final classifier, we should find a way to select the best model while avoiding overfitting. We already know the answer: cross-validation. Regression poses an additional problem: how should we evaluate our results? Accuracy is not a good idea, since we are predicting real values, it is almost impossible for us to predict exactly the final value. There are several measures that can be used (you can look at the list of functions under `sklearn.metrics` module). The most common is the **R2** score, or **coefficient of determination** that measures the proportion of the outcomes variation explained by the model, and is the default score function for regression methods in scikit-

learn. This score reaches its maximum value of 1 when the model perfectly predicts all the test target values. Using this measure, we will build a function that trains a model and evaluates its performance using five-fold cross-validation and the coefficient of determination.

```
>>> from sklearn.cross_validation import *
>>> def train_and_evaluate(clf, X_train,
y_train):
>>>     clf.fit(X_train, y_train)
>>>     print "Coefficient of determination on
training
set:",clf.score(X_train, y_train)
>>>     # create a k-fold cross validation
iterator of k=5 folds
>>>     cv = KFold(X_train.shape[0], 5,
shuffle=True,
random_state=33)
>>>     scores = cross_val_score(clf, X_train,
y_train, cv=cv)
>>>     print "Average coefficient of
determination using 5-fold
crossvalidation:",np.mean(scores)
```

First try – a linear model

The question that linear models try to answer is which hyperplane in the 14-dimensional space created by our learning features (including the target value) is located closer to them. After this hyperplane is found, prediction reduces to calculate the projection on the hyperplane of the new point, and returning the target value coordinate.

Think of our first example in [Chapter 1, Machine Learning – A Gentle Introduction](#), where we wanted to find a line separating our training instances. We could have used that line to predict the second learning attribute as a function of the first one, that is, linear regression.

But, what do we mean by closer? The usual measure is least squares: calculate the distance of each instance to the hyperplane, square it (to avoid sign problems), and sum them. The hyperplane whose sum is smaller is the least squares estimator (the hyperplane in the case if two dimensions are just a line).

Since we don't know how our data fits (it is difficult to print a 14-dimension scatter plot!), we will start with a linear model called `SGDRegressor`, which tries to minimize squared loss.

```
>>> from sklearn import linear_model
```

```
>>> clf_sgd =  
linear_model.SGDRegressor(loss='squared_loss',  
    penalty=None, random_state=42)  
>>> train_and_evaluate(clf_sgd, X_train, y_train)  
Coefficient of determination on training set:  
0.743303511411  
Average coefficient of determination using 5-  
fold crossvalidation: 0.715166411086
```

We can print the hyperplane coefficients our method has calculated, which is as follows:

```
>>> print clf_sgd.coef_  
[-0.07641527  0.06963738 -0.05935062  0.10878438  
-0.06356188  0.37260998 -0.02912886 -0.20180631  
0.08463607 -0.05534634  
-0.19521922  0.0653966 -0.36990842]
```

You probably noted the `penalty=None` parameter when we called the method. The penalization parameter for linear regression methods is introduced to avoid overfitting. It does this by penalizing those hyperplanes having some of their coefficients too large, seeking hyperplanes where each feature contributes more or less the same to the predicted value. This parameter is generally the L2 norm (the squared sums of the coefficients) or the L1 norm (that is the sum of the absolute value of the coefficients). Let's see how our model works if we introduce an L2 penalty.

```
>>> clf_sgd1 =  
linear_model.SGDRegressor(loss='squared_loss',  
    penalty='l2', random_state=42)  
>>> train_and_evaluate(clf_sgd1, X_train,  
y_train)  
Coefficient of determination on training set:  
0.743300616394  
Average coefficient of determination using 5-  
fold crossvalidation: 0.715166962417
```

In this case, we did not obtain an improvement.

Second try – Support Vector Machines for regression

The regression version of SVM can be used instead to find the hyperplane.

```
>>> from sklearn import svm  
>>> clf_svr = svm.SVR(kernel='linear')  
>>> train_and_evaluate(clf_svr, X_train,  
y_train)  
Coefficient of determination on training set:  
0.71886923342  
Average coefficient of determination using 5-  
fold crossvalidation: 0.694983285734
```

Here, we had no improvement. However, one of the main advantages of SVM is that (using what we called the kernel trick) we can use a nonlinear function, for example, a polynomial function to approximate our data.

```
>>> clf_svr_poly = svm.SVR(kernel='poly')  
>>> train_and_evaluate(clf_svr_poly, X_train,  
y_train)  
Coefficient of determination on training set:  
0.904109273301  
Average coefficient of determination using 5-  
fold cross validation: 0.754993478137
```

Now, our results are six points better in terms of coefficient of determination. We can actually improve this

by using a **Radial Basis Function (RBF)** kernel.

```
>>> clf_svr_rbf = svm.SVR(kernel='rbf')
>>> train_and_evaluate(clf_svr_rbf, X_train,
y_train)
Coefficient of determination on training set:
0.900132065979
Average coefficient of determination using 5-
fold cross validation: 0.821626135903
```

RBF kernels have been used in several problems and have shown to be very effective. Actually, RBF is the default kernel used by SVM methods in scikit-learn.

Third try – Random Forests revisited

We can try a very different approach to regression using **Random Forests**. We have previously used Random Forests for classification. When used for regression, the tree growing procedure is exactly the same, but at prediction time, when we arrive at a leaf, instead of reporting the majority class, we return a representative real value, for example, the average of the target values.

Actually, we will use **Extra Trees**, implemented in the `ExtraTreesRegressor` class within the `sklearn.ensemble` module. This method adds an extra level of randomization. It not only selects for each tree a different, random subset of features, but also randomly selects the threshold for each decision.

```
>>> from sklearn import ensemble
>>>
clf_et=ensemble.ExtraTreesRegressor(n_estimators
=10,
    compute_importances=True, random_state=42)
>>> train_and_evaluate(clf_et, X_train, y_train)
Coefficient of determination on training set:
1.0
Average coefficient of determination using 5-
fold cross validation: 0.852511952001
```

The first thing to note is that we have not only completely eliminated underfitting (achieving perfect prediction on training values), but also improved the performance by three points while using cross-validation. An interesting feature of Extra Trees is that they allow computing the importance of each feature for the regression task. Let's compute this importance as follows:

```
>>> print sort(zip(clf_et.feature_importances_,  
                   boston.feature_names), axis=0)  
  
[ ['0.000231085384564' 'AGE']  
  ['0.000909210196652' 'B']  
  ['0.00162702734638' 'CHAS']  
  ['0.00292361527201' 'CRIM']  
  ['0.00472492264278' 'DIS']  
  ['0.00489022243822' 'INDUS']  
  ['0.0067481487587' 'LSTAT']  
  ['0.00852353178943' 'NOX']  
  ['0.00873406149286' 'PTRATIO']  
  ['0.0366902590312' 'RAD']  
  ['0.0982265323415' 'RM']  
  ['0.385904111089' 'TAX']  
  ['0.439867272217' 'ZN']]
```

We can see that `ZN` (proportion of residential land zoned for lots over 25,000 sq. ft.) and `TAX` (full-value property tax rate) are by far the most influential features on our final decision.

Evaluation

As usual, let's evaluate the performance of our best method on the testing set (previously, we slightly modified our `measure_performance` function to show the coefficient of determination):

```
>>> from sklearn import metrics
>>> def measure_performance(X, y, clf,
show_accuracy=True,
    show_classification_report=True,
show_confusion_matrix=True,
    show_r2_score=False):
>>>     y_pred = clf.predict(X)
>>>     if show_accuracy:
>>>         print "Accuracy:{0:.3f}".format(
>>>             metrics.accuracy_score(y, y_pred)
>>>         ),"\n"
>>>
>>>     if show_classification_report:
>>>         print "Classification report"
>>>         print
metrics.classification_report(y, y_pred),"\n"
>>>
>>>     if show_confusion_matrix:
>>>         print "Confusion matrix"
>>>         print metrics.confusion_matrix(y,
y_pred),"\n"
>>>
>>>     if show_r2_score:
>>>         print "Coefficient of determination:
{0:.3f}".format(
```

```
>>> metrics.r2_score(y, y_pred)
>>> ) , "\n"

>>> measure_performance(X_test, y_test, clf_et,
    show_accuracy=False,
show_classification_report=False,
    show_confusion_matrix=False,
show_r2_score=True)
Coefficient of determination: 0.793
```

Once we have selected our best method and used all the available data, we could train our best method on the whole training set, but we will have no way to measure its performance on future data, simply because we do not have any more data available.

Summary

In this chapter we reviewed some of the most common supervised learning methods and some practical applications. We learned that supervised methods require instances to have both input features and a target class. In the next chapter, we will review unsupervised learning methods that do not require a target class to be learned. These methods are very useful to understand the structure of the data and can also be used as a previous step before utilizing a supervised learning model.

Chapter 3. Unsupervised Learning

Nowadays, it is a common assertion that huge amounts of data are available from the Internet for learning. If you read the previous chapters, you will see that even though supervised learning methods are very powerful in predicting future values based on the existing data, they have an obvious drawback: data must be curated; a human being should have annotated the target class for a certain number of instances. This labor is typically done by an expert (if you want to assign the correct species to iris flowers, you need somebody who knows about these flowers at least); it will probably take some time and money to complete, and it will typically not produce significant amounts of data (at least not compared with the Internet!). Every supervised learning building must stand on as much curated data as possible.

However, there are some things we can do without annotated data. Consider the case when you want to assign table seats in a wedding. You want to group people, putting similar people at the same table (the bride's family, the groom's friends, and so on). Anyone that has organized a wedding knows that this task, called

Clustering in machine learning terminology, is not an easy one. Sometimes people belong to more than one group, and you have to decide if not so similar people can be together (for example, the bride and groom's parents). Clustering involves finding groups where all elements in the group are similar, but objects in different groups are not. What does it mean to be similar is a question every clustering method must answer. The other critical question is how to separate clusters. Humans are very good at finding clusters when faced with two-dimensional data (consider identifying cities in a map just based on the presence of streets), but things become more difficult as dimensions grow.

In this chapter we will present several approximations for clustering: **k-means** (probably the most popular clustering method), **affinity propagation**, **mean shift**, and a model-based method called **Gaussian Mixture Models**.

Another example of unsupervised learning is **Dimensionality Reduction**. Suppose we represent learning instances with a large number of attributes and want to visualize them to identify their principal patterns. This is very difficult when the number of features is more than three, simply because we cannot visualize more than three dimensions. Dimensionality Reduction methods present a way to represent data points of a high

dimensional dataset in a lower dimensional space, keeping (at least partly) their pattern structure. These methods are also helpful in selecting the models we should use for learning. For example, if it is reasonable to approximate some supervised learning task using a linear hyperplane or should we resort to more complicated models.

Principal Component Analysis

Principal Component Analysis (PCA) is an orthogonal linear transformation that turns a set of possibly correlated variables into a new set of variables that are as uncorrelated as possible. The new variables lie in a new coordinate system such that the greatest variance is obtained by projecting the data in the first coordinate, the second greatest variance by projecting in the second coordinate, and so on. These new coordinates are called principal components; we have as many principal components as the number of original dimensions, but we keep only those with high variance. Each new principal component that is added to the principal components set must comply with the restriction that it should be orthogonal (that is, uncorrelated) to the remaining principal components. PCA can be seen as a method that reveals the internal structure of data; it supplies the user with a lower dimensional shadow of the original objects. If we keep only the first principal components, data dimensionality is reduced and thus it is easier to visualize the structure of data. If we keep, for example, only the first and second components, we can examine data using a two-dimensional scatter plot. As a result, PCA is useful

for exploratory data analysis before building predictive models.

For our learning methods, PCA will allow us to reduce a high-dimensional space into a low-dimensional one while preserving as much variance as possible. It is an unsupervised method since it does not need a target class to perform its transformations; it only relies on the values of the learning attributes. This is very useful for two major purposes:

- **Visualization:** Projecting a high-dimensional space, for example, into two dimensions will allow us to map our instances into a two-dimensional graph. Using these graphical visualizations, we can have insights about the distribution of instances and look at how separable instances from different classes are. In this section we will use PCA to transform and visualize a dataset.
- **Feature selection:** Since PCA can transform instances from high to lower dimensions, we could use this method to address the curse of dimensionality. Instead of learning from the original set of features, we can transform our instances with PCA and then apply a learning algorithm on top of the new feature space.

As a working example, in this section we will use a dataset of handwritten digits digitalized in matrices of 8x8 pixels, so each instance will consist initially of 64 attributes. How can we visualize the distribution of instances? Visualizing 64 dimensions at the same time is impossible for a human being, so we will use PCA to reduce the instances to two dimensions and visualize its distribution in a two-dimensional scatter graph.

We start by loading our dataset (the digits dataset is one of the sample datasets provided with scikit-learn).

```
>>> from sklearn.datasets import load_digits  
>>> digits = load_digits()  
>>> x_digits, y_digits = digits.data,  
    digits.target
```

If we print the digits keys, we get:

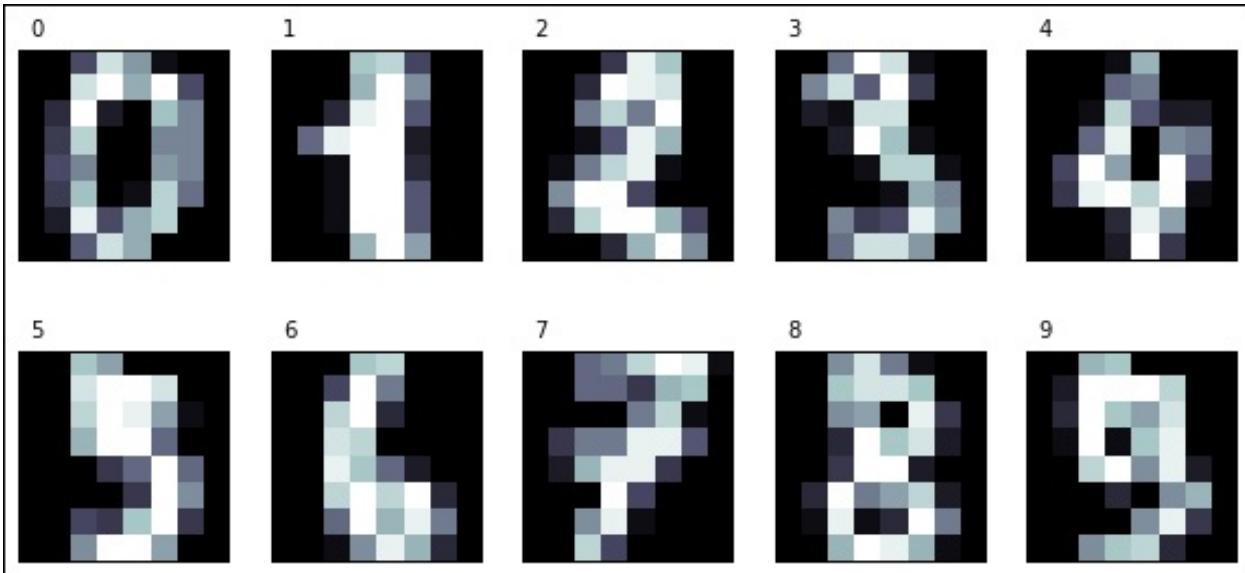
```
>>> print digits.keys()  
['images', 'data', 'target_names', 'DESCR',  
'target']
```

We will use the `data` matrix that has the instances of 64 attributes each and the `target` vector that has the corresponding digit number.

Let us print the digits to take a look at how the instances will appear:

```
>>> import matplotlib.pyplot as plt
>>> n_row, n_col = 2, 5
>>>
>>> def print_digits(images, y, max_n=10):
>>>     # set up the figure size in inches
>>>     fig = plt.figure(figsize=(2. * n_col,
2.26 * n_row))
>>>     i=0
>>>     while i < max_n and i < images.shape[0]:
>>>         p = fig.add_subplot(n_row, n_col, i
+ 1, xticks=[], yticks[])
>>>         p.imshow(images[i],
cmap=plt.cm.bone,
           interpolation='nearest')
>>>         # label the image with the target
value
>>>         p.text(0, -1, str(y[i]))
>>>         i = i + 1
>>>
>>> print_digits(digits.images, digits.target,
max_n=10)
```

These instances can be seen in the following diagram:



Define a function that will plot a scatter with the two-dimensional points that will be obtained by a PCA transformation. Our data points will also be colored according to their classes. Recall that the target class will not be used to perform the transformation; we want to investigate if the distribution after PCA reveals the distribution of the different classes, and if they are clearly separable. We will use ten different colors for each of the digits, from 0 to 9.

```
>>> def plot_pca_scatter():
>>>     colors = ['black', 'blue', 'purple',
'yellow', 'white',
        'red', 'lime', 'cyan', 'orange',
'gray']
>>>     for i in xrange(len(colors)):
```

```
>>> px = X_pca[:, 0][y_digits == i]
>>> py = X_pca[:, 1][y_digits == i]
>>> plt.scatter(px, py, c=colors[i])
>>> plt.legend(digits.target_names)
>>> plt.xlabel('First Principal Component')
>>> plt.ylabel('Second Principal Component')
```

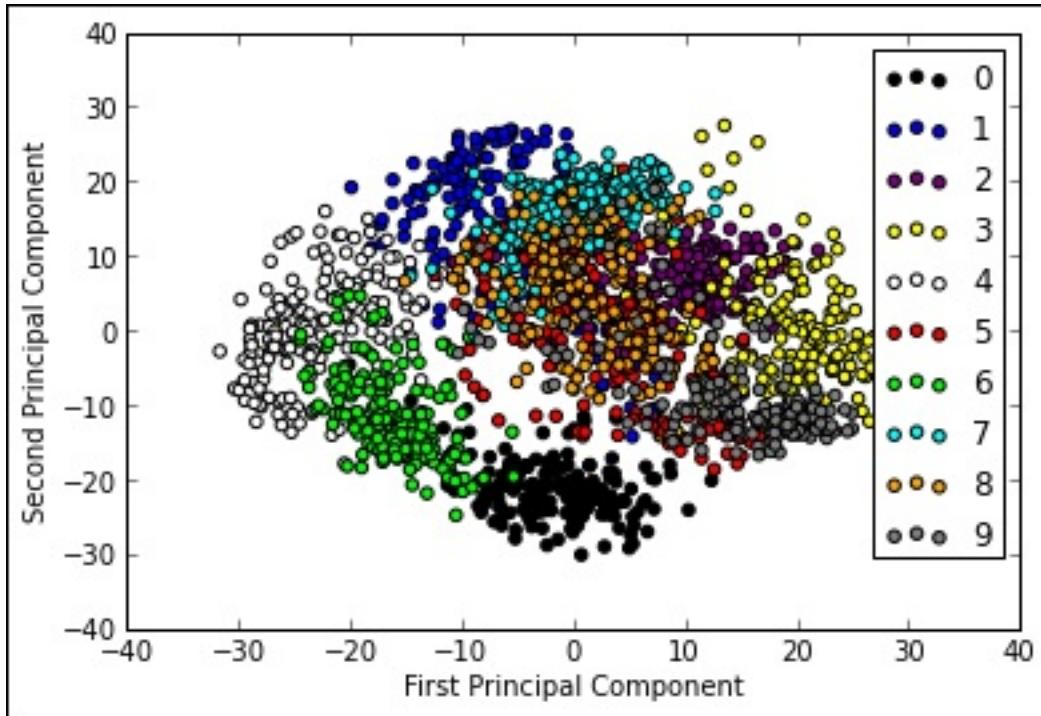
At this point, we are ready to perform the PCA transformation. In scikit-learn, PCA is implemented as a transformer object that learns n number of components through the `fit` method, and can be used on new data to project it onto these components. In scikit-learn, we have various classes that implement different kinds of PCA decompositions, such as `PCA`, `ProbabilisticPCA`, `RandomizedPCA`, and `KernelPCA`. If you need a detailed description of each, please refer to the scikit-learn documentation. In our case, we will work with the `PCA` class from the `sklearn.decomposition` module. The most important parameter we can change is `n_components`, which allows us to specify the number of features that the obtained instances will have. In our case, we want to transform instances of 64 features to instances of just ten features, so we will set `n_components` of 10.

Now we perform the transformation and plot the results:

```
>>> from sklearn.decomposition import PCA
>>> estimator = PCA(n_components=10)
>>> X_pca = estimator.fit_transform(X_digits)
```

```
>>> plot_pca_scatter()
```

The plotted results can be seen in the following diagram:



From the preceding figure, we can draw a few interesting conclusions:

- We can view the 10 different classes corresponding to the 10 digits at first sight. We see that for most classes, their instances are clearly grouped in clusters according to their target class, and also that the clusters are relatively distinct. The exception is the class corresponding to the digit **5** with instances very sparsely distributed over the plane overlap with the

other classes.

- At the other extreme, the class corresponding to the digit **0** is the most separated cluster. Intuitively, this class may be the one that is easiest to separate from the rest; that is, if we train a classifier, it should be the class with the best evaluation figures.
- Also, for topological distribution, we may predict that contiguous classes correspond to similar digits, which means they will be the most difficult to separate. For example, the clusters corresponding to digits **9** and **3** appear contiguous (which will be expected as their graphical representations are similar), so it might be more difficult to separate a **9** from a **3** than a **9** from a **4**, which is on the left-hand side, far from these clusters.

Notice that we quickly got a graph that gave us a lot of insight into the problem. This technique may be used before training a supervised classifier in order to better understand the difficulties we may encounter. With this knowledge, we may plan better feature preprocessing, feature selection, select a more suitable learning model, and so on. As we mentioned before, it can also be used to perform dimension reduction to avoid the curse of dimensionality and also may allow us to use simpler learning methods, such as linear models.

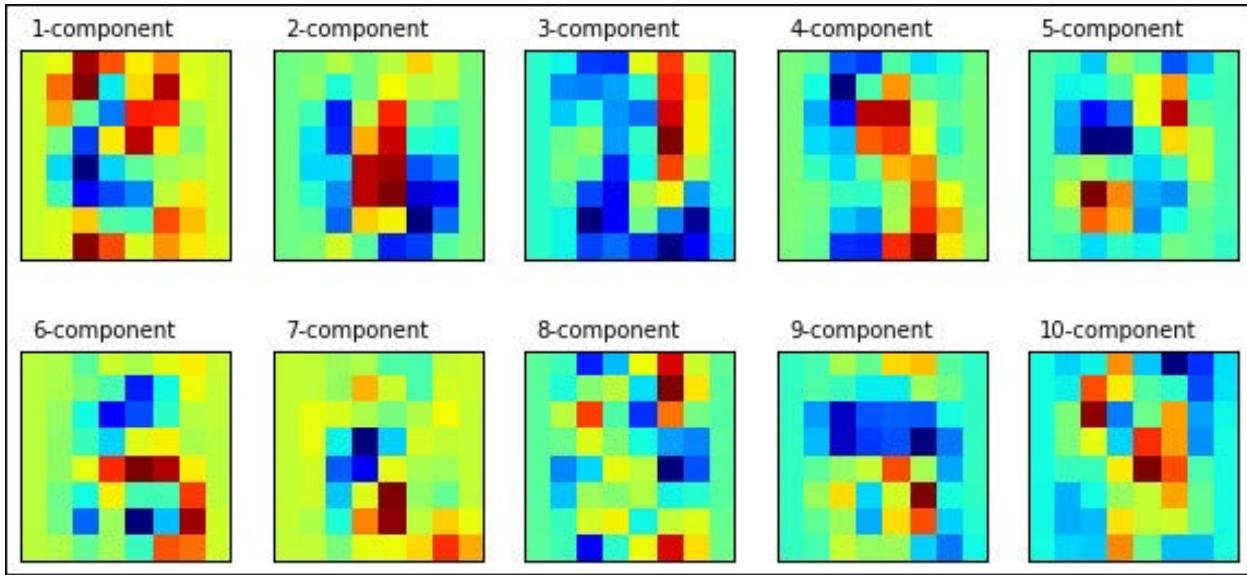
To finish, let us look at principal component transformations. We will take the principal components from the estimator by accessing the `components` attribute. Each of its components is a matrix that is used to transform a vector from the original space to the transformed space. In the scatter we previously plotted, we only took into account the first two components.

We will plot all the components in the same shape as the original data (digits).

```
>>> def print_pca_components(images, n_col,
n_row):
>>>     plt.figure(figsize=(2. * n_col, 2.26 *
n_row))
>>>     for i, comp in enumerate(images):
>>>         plt.subplot(n_row, n_col, i + 1)
>>>         plt.imshow(comp.reshape((8, 8)),
interpolation='nearest')
>>>         plt.text(0, -1, str(i + 1) + '-'
component')
>>>         plt.xticks(())
>>>         plt.yticks(())

>>>         n_components = n_row * n_col
>>>
print_pca_components(estimator.components_[:n_co
mponents], n_col, n_row)
```

The components can be seen as follows:



By taking a look at the first two components in the preceding figure, we can draw a few interesting observations:

- If you look at the second component, you can see that it mostly highlights the central region of the image. The digit class that is most affected by this pattern is **0**, since its central region is empty. This intuition is confirmed by looking at our previous scatter plot. If you look at the cluster corresponding to the digit **0**, you can see it is the one that has the lower values for the second component.
- Regarding the first component, as we see in the scatter plot, it is very useful to separate the clusters corresponding to the digit **4** (extreme left, low value)

and the digit **3** (extreme right, high value). If you see the first component plot, it agrees with this observation. You can see that the regions corresponding to the zone are very similar to the digit **3**, while it has color in the zones that are characteristic of the digit **4**.

If we used additional components, we will get more characteristics to be able to separate the classes into new dimensions. For example, we could add the third principal component and try to plot our instances in a tridimensional scatter plot.

In the next section, we will show another unsupervised group of methods: clustering algorithms. Like dimensionality-reduction algorithms, clustering does not need to know a target class. However, clustering methods try to group instances, looking for those that are (in some way) similar. We will see, however, that clustering methods, like supervised methods, can use PCA to better visualize and analyze their results.

Clustering handwritten digits with k-means

K-means is the most popular clustering algorithm, because it is very simple and easy to implement and it has shown good performance on different tasks. It belongs to the class of partition algorithms that simultaneously partition data points into distinct groups called **clusters**. An alternative group of methods, which we will not cover in this book, are hierarchical clustering algorithms. These find an initial set of clusters and divide or merge them to form new ones.

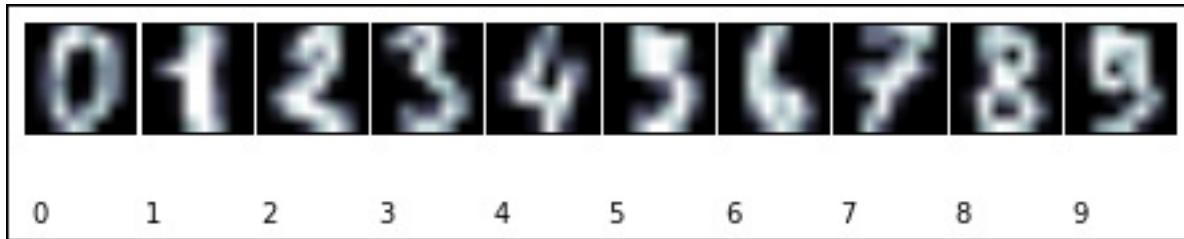
The main idea behind k-means is to find a partition of data points such that the squared distance between the cluster mean and each point in the cluster is minimized. Note that this method assumes that you know *a priori* the number of clusters your data should be divided into.

We will show in this section how k-means works using a motivating example, the problem of clustering handwritten digits. So, let us first import our dataset into our Python environment and show how handwritten digits look (we will use a slightly different version of the `print_digits` function we introduced in the previous

section).

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> from sklearn.datasets import load_digits
>>> from sklearn.preprocessing import scale
>>> digits = load_digits()
>>> data = scale(digits.data)
>>>
>>> def print_digits(images,y,max_n=10):
>>>     # set up the figure size in inches
>>>     fig = plt.figure(figsize=(12, 12))
>>>     fig.subplots_adjust(left=0, right=1,
bottom=0, top=1,
           hspace=0.05, wspace=0.05)
>>>     i = 0
>>>     while i <max_n and i <images.shape[0]:
>>>         # plot the images in a matrix of
20x20
>>>         p = fig.add_subplot(20, 20, i + 1,
xticks=[],
           yticks[])
>>>         p.imshow(images[i],
cmap=plt.cm.bone)
>>>         # label the image with the target
value
>>>         p.text(0, 14, str(y[i]))
>>>         i = i + 1
>>>
>>> print_digits(digits.images, digits.target,
max_n=10)
```

The print digits can be seen in the following:



You can see that the dataset contains the corresponding number associated as a target class, but since we are clustering we will not use this information until evaluation time. We will just see if we can group the figures based on their similarity, and form the ten clusters we can expect.

As usual, we must separate train and testing sets as follows:

```
>>> from sklearn.cross_validation import  
train_test_split  
>>> X_train, X_test, y_train, y_test,  
images_train,  
    images_test = train_test_split(  
        data, digits.target, digits.images,  
test_size=0.25,  
        random_state=42)  
>>>  
>>> n_samples, n_features = X_train.shape  
>>> n_digits = len(np.unique(y_train))
```

```
>>> labels = y_train
```

Once we have our training set, we are ready to cluster instances. What the k-means algorithm does is:

1. Select an initial set of cluster centers at random.
2. Find the nearest cluster center for each data point, and assign the data point closest to that cluster.
3. Compute the new cluster centers, averaging the values of the cluster data points, and repeat until cluster membership stabilizes; that is, until a few data points change their clusters after each iteration.

Because of how k-means works, it can converge to local minima, and the initial set of cluster centers could greatly affect the clusters found. The usual approach to mitigate this is to try several initial sets and select the set with minimal value for the sum of squared distances between cluster centers (or inertia). The implementation of k-means in scikit-learn already does this (the `n_init` parameter allows us to establish how many different centroid configurations the algorithm will try). It also allows us to specify that the initial centroids will be sufficiently separated, leading to better results. Let's see how this works on our dataset.

```
>>> from sklearn import cluster  
>>> clf = Cluster.KMeans(init='k-means++',
```

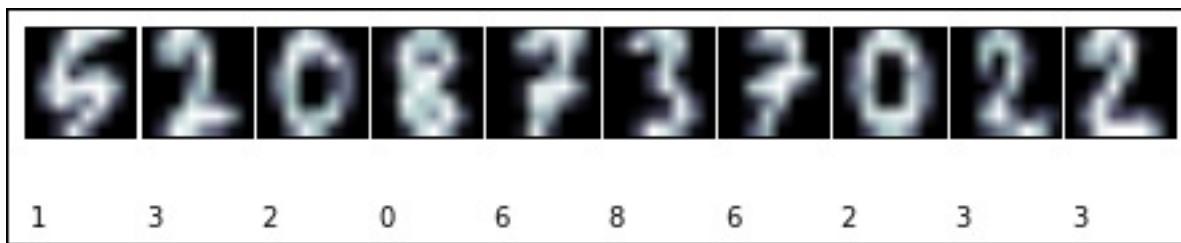
```
n_clusters=10, random_state=42)  
>>> clf.fit(X_train)
```

The procedure is similar to the one used for supervised learning, but note that the `fit` method only takes the training data as an argument. Also observe that we need to specify the number of clusters. We can perceive this number because we know that clusters represent numbers.

If we print the value of the `labels_` attribute of the classifier, we get a list of the cluster numbers associated to each training instance.

```
>>> print_digits(images_train, clf.labels_,  
max_n=10)
```

The cluster can be seen in the following diagram:



Note that the cluster number has nothing to do with the real number value. Remember that we have not used the class to classify; we only grouped images by similarity. Let's see how our algorithm behaves on the testing data.

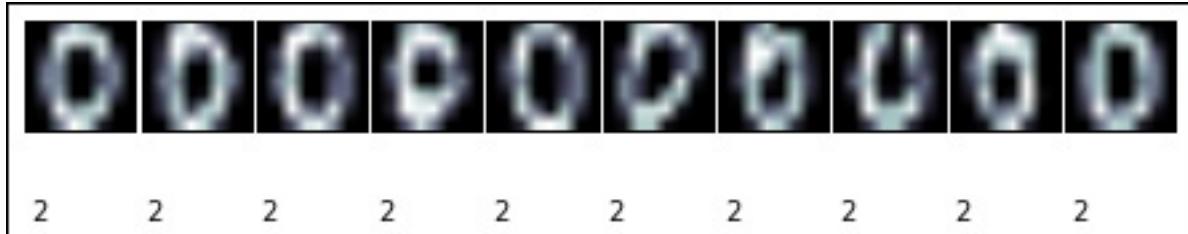
To predict the clusters for training data, we use the usual `predict` method of the classifier.

```
>>> y_pred=clf.predict(X_test)
```

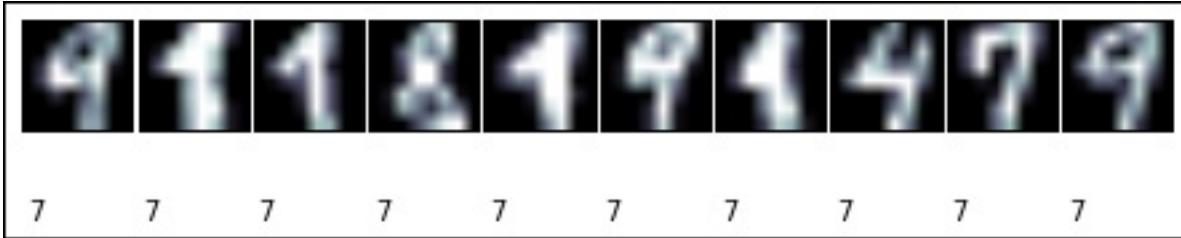
Let us see how clusters look:

```
>>> def print_cluster(images, y_pred,
cluster_number) :
>>>     images = images[y_pred==cluster_number]
>>>     y_pred = y_pred[y_pred==cluster_number]
>>>     print_digits(images, y_pred,max_n=10)
>>> for i in range(10):
>>>     print_cluster(images_test, y_pred, i)
```

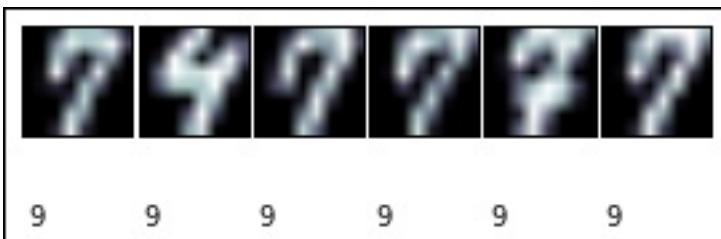
This code shows ten images from each cluster. Some clusters are very clear, as shown in the following figure:



Cluster number **2** corresponds to zeros. What about cluster number **7**?



It is not so clear. It seems cluster 7 is something like drawn numbers that look similar to the digit nine. Cluster number 9 only has six instances, as shown in the following figure:



It must be clear after reading that we are not classifying images here (as in the face examples in the previous chapter). We are grouping into ten classes (you can try changing the number of clusters and see what happens).

How can we evaluate our performance? Precision and all that stuff does not work, since we have no target classes to compare with. To evaluate, we need to know the "real"

clusters, whatever that means. We can suppose, for our example, that each cluster includes every drawing of a certain number, and only that number. Knowing this, we can compute the **adjusted Rand index** between our cluster assignment and the expected one. The Rand index is a similar measure for accuracy, but it takes into account the fact that classes can have different names in both assignments. That is, if we change class names, the index does not change. The adjusted index tries to deduct from the result coincidences that have occurred by chance. When you have the exact same clusters in both sets, the Rand index equals one, while it equals zero when there are no clusters sharing a data point.

```
>>> from sklearn import metrics  
>>> print "Adjusted rand score:  
  
{:.2}{}".format(metrics.adjusted_rand_score(y_test  
, y_pred))  
Adjusted rand score:0.57
```

We can also print the confusion matrix as follows:

```
>>> print metrics.confusion_matrix(y_test,  
y_pred)  
[[ 0  0 43  0  0  0  0  0  0  0]  
 [20  0  0  7  0  0  0 10  0  0]  
 [ 5  0  0 31  0  0  0  1  1  0]  
 [ 1  0  0  1  0  1  4  0 39  0]  
 [ 1 50  0  0  0  0  1  2  0  1]]
```

```
[ 1  0  0  0  1  41  0  0  16  0]
[ 0  0  1  0  44  0  0  0  0  0]
[ 0  0  0  0  0  1  34  1  0  5]
[21  0  0  0  0  3  1  2  11  0]
[ 0  0  0  0  0  2  3  3  40  0] ]
```

Observe that the class 0 in the test set (which coincides with number 0 drawings) is completely assigned to the cluster number 2. We have problems with number 8: 21 instances are assigned class 0, while 11 are assigned class 8, and so on. Not so good after all.

If we want to graphically show how k-means clusters look like, we must plot them on a two-dimensional plane. We have learned how to do that in the previous section: Principal Component Analysis (PCA). Let's construct a **meshgrid** of points (after dimensionality reduction), calculate their assigned cluster, and plot them.

Note

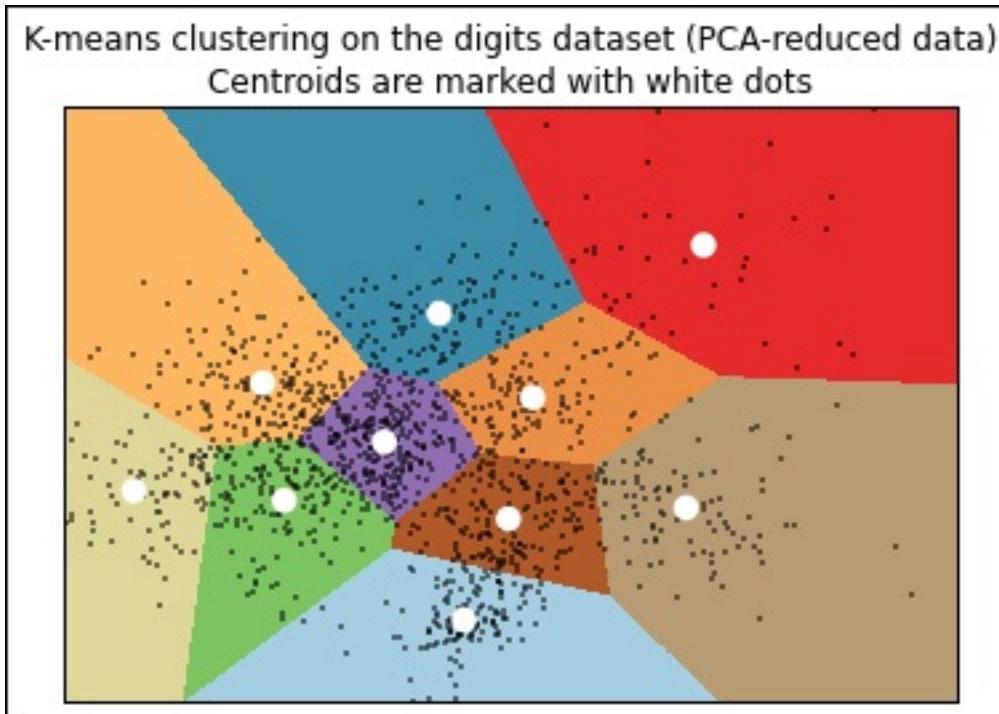
This example is taken from the very nice scikit-learn tutorial at <http://scikit-learn.org/>.

```
>>> from sklearn import decomposition
>>> pca =
decomposition.PCA(n_components=2).fit(X_train)
>>> reduced_X_train = pca.transform(X_train)
>>> # Step size of the mesh.
>>> h = .01
```

```
>>> # point in the mesh [x_min, m_max]x[y_min,
y_max].
>>> x_min, x_max = reduced_X_train[:, 0].min() +
1,
    reduced_X_train[:, 0].max() - 1
>>> y_min, y_max = reduced_X_train[:, 1].min() +
1,
    reduced_X_train[:, 1].max() - 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max,
h),
    np.arange(y_min, y_max, h))
>>> kmeans = cluster.KMeans(init='k-means++',
n_clusters=n_digits,
    n_init=10)
>>> kmeans.fit(reduced_X_train)
>>> Z = kmeans.predict(np.c_[xx.ravel(),
yy.ravel()])
>>> # Put the result into a color plot
>>> Z = Z.reshape(xx.shape)
>>> plt.figure(1)
>>> plt.clf()
>>> plt.imshow(Z, interpolation='nearest',
extent=(xx.min(), xx.max(), yy.min(),
yy.max()), cmap=plt.cm.Paired,
aspect='auto', origin='lower')
>>> plt.plot(reduced_X_train[:, 0],
reduced_X_train[:, 1], 'k.',
    markersize=2)
>>> # Plot the centroids as a white X
>>> centroids = kmeans.cluster_centers_
>>> plt.scatter(centroids[:, 0], centroids[:, 1], marker='.',
    s=169, linewidths=3, color='w', zorder=10)
>>> plt.title('K-means clustering on the digits')
```

```
dataset (PCA  
    reduced data)\nCentroids are marked with  
white dots')  
>>> plt.xlim(x_min, x_max)  
>>> plt.ylim(y_min, y_max)  
>>> plt.xticks()  
>>> plt.yticks()  
>>> plt.show()
```

The k-means clustering on the digits dataset can be seen in the following diagram:



Alternative clustering methods

The scikit-learn toolkit includes several clustering algorithms, all of them including similar methods and parameters to those we used in k-means. In this section we will briefly review some of them, suggesting some of their advantages.

A typical problem for clustering is that most methods require the number of clusters we want to identify. The general approach to solve this is to try different numbers and let an expert determine which works best using techniques such as dimensionality reduction to visualize clusters. There are also some methods that try to automatically calculate the number of clusters. Scikit-learn includes an implementation of **Affinity Propagation**, a method that looks for instances that are the most representative of others, and uses them to describe the clusters. Let's see how it works on our digit-learning problem:

```
>>> aff = cluster.AffinityPropagation()  
>>> aff.fit(X_train)  
>>> print aff.cluster_centers_.shape  
(112,)
```

Affinity propagation detected 112 clusters in our training set. It seems, after all, that the numbers are not so similar between them. You can try drawing the clusters using the `print_digits` function, and see which clusters seemed to group. The `cluster_centers_indices_` attribute represents what Affinity Propagation found as the canonical elements of each cluster.

Another method that calculates cluster number is `MeanShift()`. If we apply it to our example, it detects 18 clusters as follows:

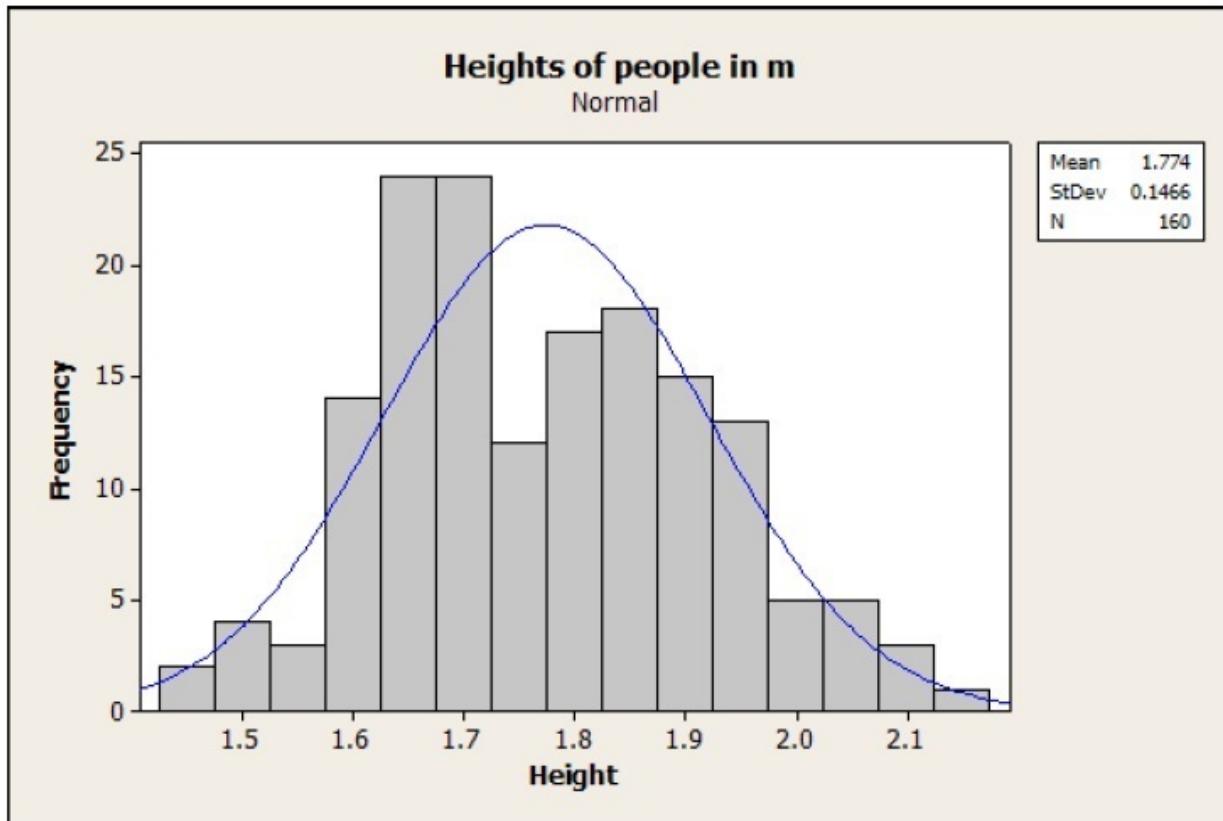
```
>>> ms = cluster.MeanShift()  
>>> ms.fit(X_train)  
>>> print ms.cluster_centers_.shape  
(18, 64)
```

In this case, the `cluster_centers_` attribute shows the hyperplane cluster centroids. The two previous examples show that results can vary a lot depending on the method we are using. Which clustering method to use depends on the problem we are solving and the type of clusters we want to find.

Note that, for the last two methods, we cannot use the Rand score to evaluate performance because we do not have a canonical set of clusters to compare with. We can, however, measure the inertia of the clustering, since

inertia is the sum of distances from each data point to the centroid; we expect near-zero numbers. Unfortunately, there is currently no way in scikit-learn to measure inertia except for the k-means method.

Finally, we will try a probabilistic approach to clustering, using **Gaussian Mixture Models (GMM)**. We will see, from a procedural view, that it is very similar to k-means, but their theoretical principles are quite different. GMM assumes that data comes from a mixture of finite Gaussian distributions with unknown parameters. A Gaussian distribution is a well-known distribution function within statistics used to model many phenomena. It has a bell shaped function centered in the mean value; you have probably seen the following drawing before:



If we take a sufficiently large sample of men and measure their height, the histogram (proportion of men with each specific height) can be adjusted by a Gaussian distribution with mean 1.774 meters and standard deviation of 0.1466 meters. Mean indicates the most probable value (which coincides with the peak of the curve), and standard deviation indicates how spread out the results are; that is, how far they can appear from the mean values. If we measure the area beneath the curve (that is, its integral) between two specific heights, we can know, given a man,

how probable it is that his height lies between the two values, in case the distribution is correct. Now, why should we expect that distribution and not another?

Actually, not every phenomenon has the same distribution, but a theorem called the **Central Limit**

Theorem tells us that whenever we repeat an experiment a large number of times (for example, measuring some people heights), the distribution of the average results can be approximated by a Gaussian.

Generally, we have a multivariate (that is, involving more than one feature) distribution, but the idea is the same.

There is a point in the hyperplane (the mean) most instances will be closer to; when we move away from the mean, the probability of finding a point in the cluster will decrease. How far this probability decreases is dependent on the second parameter, the variance. As we said, GMM assumes each cluster has a multivariate normal distribution, and the method objective is to find the k centroids (estimating mean and variance from training data using an algorithm called **Expectation-Maximization (EM)**) and assign each point to the nearest mean. Let's see how it works on our example.

```
>>> from sklearn import mixture  
>>> gm = mixture.GMM(n_components=n_digits,  
covariance_type='tied', random_state=42)  
>>> gm.fit(X_train)
```

```
GMM(covariance_type='tied', init_params='wmc',
min_covar=0.001, n_components=10, n_init=1,
n_iter=100,
params='wmc', random_state=42, thresh=0.01)
```

You can observe that the procedure is exactly the same as the one we used for k-means. `covariance_type` is a method parameter that indicates how we expect features; that is, each pixel to be related. For example, we can suppose that they are independent, but we can also expect that closer points are correlated, and so on. For the moment, we will use the tied covariance type. In the next chapter, we will show some techniques to select between different parameter values.

Let's see how it performs on our testing data:

```
>>> # Print train clustering and confusion
matrix
>>> y_pred = gm.predict(X_test)
>>> print "Adjusted rand
score:
{:.2}""".format(metrics.adjusted_rand_score(y_test
,
y_pred))
Adjusted rand score:0.65

>>> print "Homogeneity score:{:.2}
".format(metrics.homogeneity_score(y_test,
y_pred))
Homogeneity score:0.74
```

```
>>> print "Completeness score: {:.2f}\n    ".format(metrics.completeness_score(y_test,\n    y_pred))\nCompleteness score: 0.79
```

Compared to k-means, we achieved a better Rand score (0.65 versus 0.59), indicating that we have better aligned our clusters with the original digits. We also included two interesting measures included in `sklearn.metrics`.

Homogeneity is a number between 0.0 and 1.0 (greater is better). A value of 1.0 indicates that clusters only contain data points from a single class; that is, clusters effectively group similar instances. **Completeness**, on the other hand, is satisfied when every data point of a given class is within the same cluster (meaning that we have grouped all possible instances of the class, instead of building several uniform but smaller clusters). We can see homogeneity and completeness as the unsupervised versions of precision and recall.

Summary

In this chapter we presented some of the most important unsupervised learning methods. We did not intend to provide you with an exhaustive introduction to all the possible methods, but instead a brief introduction to these kinds of techniques. We described how we can use unsupervised algorithms to perform a quick data analysis to understand the behavior of the dataset and also perform dimensionality reduction. Both applications are very useful as a step before applying a supervised learning method. We also applied unsupervised learning techniques such as k-means to resolve problems without using a target class—a very useful way to create applications on top of untagged data.

In [Chapter 4](#), *Advanced Features*, we will look at techniques that will allow us to obtain better results in the application of machine learning algorithms. We will look at data-preprocessing and feature-selection techniques to obtain better features to learn from. Also, we will use grid search techniques to obtain the parameters that produce the best performance with our algorithms.

Chapter 4. Advanced Features

In the previous chapters we have studied several algorithms for very different tasks, from classification and regression to clustering and dimensionality reduction. We showed how we can apply these algorithms to predict results when faced with new data. That is what machine learning is all about. In this last chapter, we want to show some important concepts and methods you should take into account if you want to do real-world machine learning.

- In real-world problems, usually data is not already expressed by attribute/float value pairs, but through more complex structures or is not structured at all. We will learn **feature extraction** techniques that will allow us to extract scikit-learn features from data.
- From the initial set of available features, not all of them will be useful for our algorithms to learn from; in fact, some of them may degrade our performance. We will address the problem of selecting the most adequate feature set, a process known as **feature selection**.
- Finally, as we have seen in the examples in this

book, many of the machine learning algorithms have parameters that must be set in order to use them. To do that, we will review **model selection** techniques; that is, methods to select the most promising hyperparameters to our algorithms.

All these steps are crucial in order to obtain decent results when working with machine learning applications.

Feature extraction

The usual scenario for learning tasks such as those presented in this book include a list of instances (represented as feature/value pairs) and a special feature (the target class) that we want to predict for future instances based on the values of the remaining features. However, the source data does not usually come in this format. We have to extract what we think are potentially useful features and convert them to our learning format. This process is called feature extraction or feature engineering, and it is an often underestimated but very important and time-consuming phase in most real-world machine learning tasks. We can identify two different steps in this task:

- **Obtain features:** This step involves processing the source data and extracting the learning instances, usually in the form of feature/value pairs where the value can be an integer or float value, a string, a categorical value, and so on. The method used for extraction depends heavily on how the data is presented. For example, we can have a set of pictures and generate an integer-valued feature for each pixel, indicating its color level, as we did in the face recognition example in [Chapter 2, Supervised](#)

Learning. Since this is a very task-dependent job, we will not delve into details and assume we already have this setting for our examples.

- **Convert features:** Most scikit-learn algorithms assume as an input a set of instances represented as a list of float-valued features. How to get these features will be the main subject of this section.

We can, as we did in [Chapter 2](#), *Supervised Learning*, build ad hoc procedures to convert the source data. There are, however, tools that can help us to obtain a suitable representation. The Python package pandas (<http://pandas.pydata.org/>), for example, provides data structures and tools for data analysis. It aims to provide similar features to those of R, the popular language and environment for statistical computing. We will use pandas to import the Titanic data we presented in [Chapter 2](#), *Supervised Learning*, and convert them to the scikit-learn format.

Let's start by importing the original `titanic.csv` data into a pandas `DataFrame` data structure (`DataFrame` is essentially a two-dimensional labeled data structure where columns can potentially include different data types and each row represents an instance). As usual, we previously import the `numpy` and `pyplot` packages.

```
>>> %pylab inline  
>>> import pandas as pd  
>>> import numpy as np  
>>> import matplotlib.pyplot as plt
```

Then we import the Titanic data with pandas.

```
>>> titanic = pd.read_csv('data/titanic.csv')  
>>> print titanic  
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 1313 entries, 0 to 1312  
Data columns (total 11 columns):  
row.names    1313 non-null values  
pclass       1313 non-null values  
survived     1313 non-null values  
name         1313 non-null values  
age          633 non-null values  
embarked     821 non-null values  
home.dest    754 non-null values  
room         77 non-null values  
ticket        69 non-null values  
boat          347 non-null values  
sex          1313 non-null values  
dtypes: float64(1), int64(2), object(8)
```

You can see that each csv column has a corresponding feature into the DataFrame, and that the feature type is induced from the available data. We can inspect some features to see what they look like.

```
>>> print titanic.head() [[ 'pclass', 'survived',  
'age', 'embarked',
```

```

    'boat', 'sex']]]

pclass      survived       age      embarked      boat
sex
0      1st           1   29.0000  Southampton      2
female
1      1st           0   2.0000  Southampton     NaN
female
2      1st           0   30.0000  Southampton  (135)
male
3      1st           0   25.0000  Southampton     NaN
female
4      1st           1   0.9167  Southampton      11
male

```

The main difficulty we have now is that scikit-learn methods expect real numbers as feature values. In [Chapter 2, Supervised Learning](#), we used the `LabelEncoder` and `OneHotEncoder` preprocessing methods to manually convert certain categorical features into 1-of-K values (generating a new feature for each possible value; valued 1 if the original feature had the corresponding value and 0 otherwise). This time, we will use a similar scikit-learn method, `DictVectorizer`, which automatically builds these features from the different original feature values. Moreover, we will program a method to encode a set of columns in a unique step.

```

>>> from sklearn import feature_extraction
>>> def one_hot_dataframe(data, cols,
replace=False):

```

```

>>>     vec =
feature_extraction.DictVectorizer()
>>>     mkdict = lambda row: dict((col,
row[col]) for col in cols)
>>>     vecData =
pd.DataFrame(vec.fit_transform(
>>>         data[cols].apply(mkdict,
axis=1)).toarray())
>>>     vecData.columns =
vec.get_feature_names()
>>>     vecData.index = data.index
>>>     if replace:
>>>         data = data.drop(cols, axis=1)
>>>         data = data.join(vecData)
>>>     return (data, vecData)

```

The `one_hot_dataframe` method (based on the script at <https://gist.github.com/kljensen/5452382>) takes a pandas `DataFrame` data structure and a list of columns and encodes each column into the necessary 1-of-K features. If the `replace` parameter is `True`, it will also substitute the original column with the new set. Let's see it applied to the categorical `pclass`, `embarked`, and `sex` features (`titanic_n` only contains the previously created columns):

```

>>> titanic,titanic_n =
one_hot_dataframe(titanic, ['pclass',
    'embarked', 'sex'], replace=True)
>>> titanic.describe()
<class 'pandas.core.frame.DataFrame'>

```

```
Index: 8 entries, count to max
Data columns (total 12 columns):
row.names           8 non-null values
survived            8 non-null values
age                 8 non-null values
embarked            8 non-null values
embarked=Cherbourg 8 non-null values
embarked=Queenstown 8 non-null values
embarked=Southampton 8 non-null values
pclass=1st          8 non-null values
pclass=2nd          8 non-null values
pclass=3rd          8 non-null values
sex=female          8 non-null values
sex=male             8 non-null values
dtypes: float64(12)
```

The `pclass` attribute has been converted to three `pclass=1st`, `pclass=2nd`, `pclass=3rd` features, and similarly for the other two features. Note that the `embarked` feature has not disappeared, This is due to the fact that the original `embarked` attribute included `NaN` values, indicating a missing value; in those cases, every feature based on `embarked` will be valued `0`, but the original feature whose value is `NaN` remains, indicating the feature is missing for certain instances. Next, we encode the remaining categorical attributes:

```
>>> titanic, titanic_n =
one_hot_dataframe(titanic, ['home.dest',
    'room', 'ticket', 'boat'], replace=True)
```

We also have to deal with missing values, since `DecisionTreeClassifier` we plan to use does not admit them on input. Pandas allow us to replace them with a fixed value using the `fillna` method. We will use the mean age for the `age` feature, and 0 for the remaining missing attributes.

```
>>> mean = titanic['age'].mean()  
>>> titanic['age'].fillna(mean, inplace=True)  
>>> titanic.fillna(0, inplace=True)
```

Now, all of our features (except for `Name`) are in a suitable format. We are ready to build the test and training sets, as usual.

```
>>> from sklearn.cross_validation import  
train_test_split  
>>> titanic_target = titanic['survived']  
>>> titanic_data = titanic.drop(['name',  
'row.names', 'survived'],  
axis=1)  
>>> X_train, X_test, y_train, y_test =  
    train_test_split(titanic_data,  
titanic_target, test_size=0.25,  
random_state=33)
```

We decided to simply drop the `name` attribute, since we do not expect it to be informative about the survival status (we have one different value for each instance, so we can generalize over it). We also specified the `survived`

feature as the target class, and consequently eliminated it from the training vector.

Let's see how a decision tree works with the current feature set.

```
>>> from sklearn import tree
>>> dt =
tree.DecisionTreeClassifier(criterion='entropy')
>>> dt = dt.fit(X_train, y_train)
>>> from sklearn import metrics
>>> y_pred = dt.predict(X_test)
>>> print "Accuracy:
{0:.3f}""".format(metrics.accuracy_score(y_test,
y_pred)), "\n"
Accuracy:0.839
```

Feature selection

Until now, when training our decision tree, we used every available feature in our learning dataset. This seems perfectly reasonable, since we want to use as much information as there is available to build our model. There are, however, two main reasons why we would want to restrict the number of features used:

- Firstly, for some methods, especially those (such as decision trees) that reduce the number of instances used to refine the model at each step, it is possible that irrelevant features could suggest correlations between features and target classes that arise just by chance and do not correctly model the problem. This aspect is also related to overfitting; having certain over-specific features may lead to poor generalization. Besides, some features may be highly correlated, and will simply add redundant information.
- The second reason is a real-world one. A large number of features could greatly increase the computation time without a corresponding classifier improvement. This is of particular importance when working with Big Data, where the number of instances and features could easily grow to several

thousand or more. Also, in relation to the curse of dimensionality, learning a generalizable model from a dataset with too many features relative to the number of instances can be difficult.

As a result, working with a smaller feature set may lead to better results. So we want to find some way to algorithmically find the best features. This task is called feature selection and is a crucial step when we aim to get decent results with machine learning algorithms. If we have poor features, our algorithm will return poor results no matter how sophisticated our machine learning algorithm is.

Consider, for example, our very simple Titanic example. We started with just 11 features, but after 1-of-K encoding they grew to 581.

```
>>> print titanic
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1313 entries, 0 to 1312 Columns: 581
entries, row.names to ticket=L15 1s dtypes:
float64(578), int64(2), object(1)
```

This does not pose an important computational problem, but consider what could happen if, as previously demonstrated, we represent each document in a dataset as the number of occurrences of each possible word.

Another problem is that decision trees suffer from overfitting. If branching is based on a very small number of instances, the prediction power of the built model will decrease on future data. One solution to this is to adjust model parameters (such as the maximum tree depth or the minimum required number of instances at a leaf node). In this example, however, we will take a different approach: we will try to limit the features to the most relevant ones.

What do we mean by relevant? This is an important question. A general approach is to find the smallest set of features that correctly characterize the training data. If a feature always coincides with the target class (that is, it is a perfect predictor), it is enough to characterize the data. On the other hand, if a feature always has the same value, its prediction power will be very low.

The general approach in feature selection is to get some kind of evaluation function that, when given a potential feature, returns a score of how useful the feature is, and then keeps the features with the highest scores. These methods may have the disadvantage of not detecting correlations between features. Other methods may be more brute force: try all possible subsets of the original feature list, train the algorithm on each combination, and keep the combination that gets the best results.

As an evaluation method, we can, for instance, use a statistical test that measures how probable it is that two random variables (say, a given feature and the target class) are independent; that is, there is no correlation between them.

Scikit-learn provides several methods in the `feature_selection` module. We will use the `SelectPercentile` method that, when given a statistical test, selects a user-specified percentile of features with the highest scoring. The most popular statistical test is the χ^2 (chi-squared) statistic. Let's see how it works for our Titanic example; we will use it to select 20 percent of the most important features:

```
>>> from sklearn import feature_selection  
>>> fs = feature_selection.SelectPercentile(  
        feature_selection.chi2, percentile=20)  
>>> X_train_fs = fs.fit_transform(X_train,  
y_train)
```

The `X_train_fs` array now has the statistically more important features. We can now train our decision tree on this data.

```
>>> dt.fit(X_train_fs, y_train)  
>>> X_test_fs = fs.transform(X_test)  
>>> y_pred_fs = dt.predict(X_test_fs)  
>>> print "Accuracy:"
```

```
{0:.3f}").format(metrics.accuracy_score(y_test,  
y_pred_fs)), "\n"  
Accuracy: 0.845
```

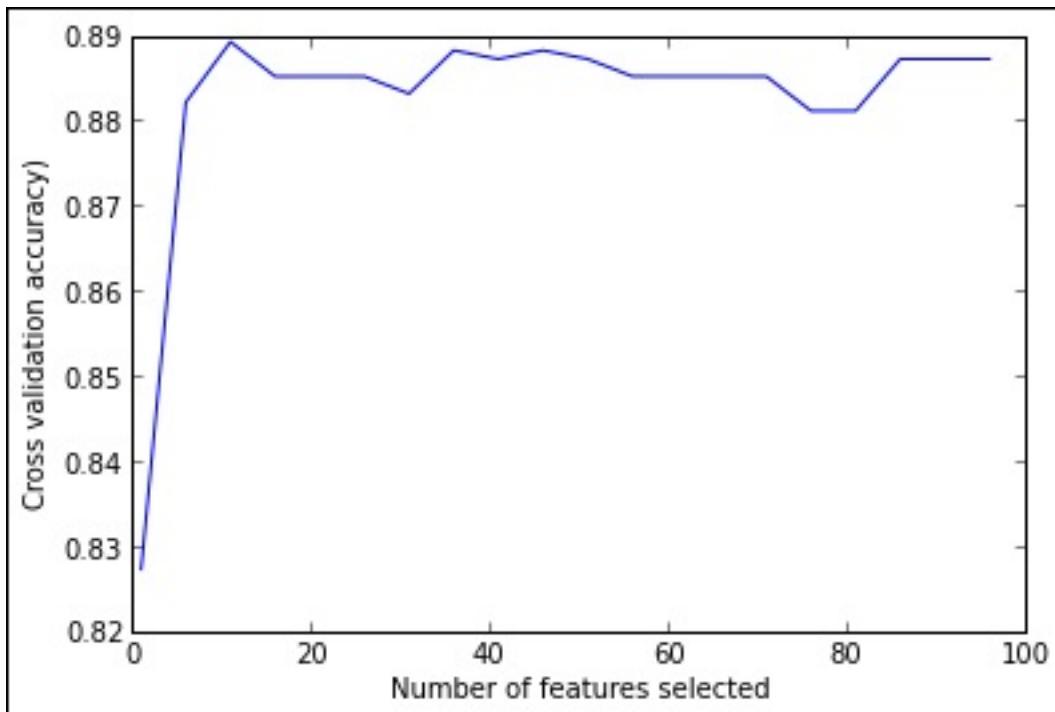
We can see that the accuracy on the training set improved half a point after feature selection on the training set.

Is it possible to find the optimal number of features? If by optimal we mean with the best performance on the training set, it is actually possible; we can simply use a brute-force approach and try with different numbers of features while measuring their performance on the training set using cross-validation.

```
>>> from sklearn import cross_validation  
>>>  
>>> percentiles = range(1, 100, 5)  
>>> results = []  
>>> for i in range(1,100,5):  
>>>     fs = feature_selection.SelectPercentile(  
            feature_selection.chi2, percentile=i  
        )  
>>>     X_train_fs = fs.fit_transform(X_train,  
y_train)  
>>>     scores =  
cross_validation.cross_val_score(dt, X_train_fs,  
y_train, cv=5)  
>>>     results = np.append(results,  
scores.mean())  
>>> optimal_percentile = np.where(results ==  
results.max())[0]  
>>> print "Optimal number of features:"
```

```
{0}'.format(  
    percentiles[optimal_percentile]), '\n'  
Optimal number of features:11  
>>>  
>>> # Plot number of features VS. cross-  
validation scores  
>>> import pylab as pl  
>>> pl.figure()  
>>> pl.xlabel("Number of features selected")  
>>> pl.ylabel("Cross-validation accuracy")  
>>> pl.plot(percentiles, results)
```

The following figure shows how cross-validation accuracy changes with the number of features:



We can see that accuracy quickly improves when we start adding features, remaining stable after the percentile of features turns about 10. In fact, the best accuracy is achieved when using 64 of the original 581 features (at the 11 percent percentile). Let's see if this actually improved performance on the testing set.

```
>>> fs = feature_selection.SelectPercentile(  
        feature_selection.chi2,  
  
percentile=percentiles[optimal_percentile])  
>>> X_train_fs = fs.fit_transform(X_train,  
y_train)  
>>> dt.fit(X_train_fs, y_train)  
>>> X_test_fs = fs.transform(X_test)  
>>> y_pred_fs = dt.predict(X_test_fs)  
>>> print "Accuracy:  
{0:.3f}".format(metrics.accuracy_score(y_test,  
    y_pred_fs)), "\n"  
Accuracy:0.848
```

The performance improved slightly, again. Compared with our initial performance, we have finally improved by almost one accuracy point using only 11 percent of the features.

The reader may have noted that while creating our classifier, we used the default parameters, except for the splitting criterion, where we have used `entropy`. Can we improve our model using different parameters? This task

is called model selection, and we will address it in detail in the next section using a different learning example. For now, let's just test if the alternative method (`gini`) would result in better performance for our example. To do this, we will again use cross-validation.

```
>>> dt =
tree.DecisionTreeClassifier(criterion='entropy')
>>> scores =
cross_validation.cross_val_score(dt, X_train_fs,
    y_train, cv=5)
>>> print "Entropy criterion accuracy on
    cv: {:.3f}".format(scores.mean())
Entropy criterion accuracy on cv: 0.889
>>> dt =
tree.DecisionTreeClassifier(criterion='gini')
>>> scores =
cross_validation.cross_val_score(dt, X_train_fs,
    y_train, cv=5)
>>> print "Gini criterion accuracy on
    cv: {:.3f}".format(scores.mean())
Gini criterion accuracy on cv: 0.897
```

The Gini criterion performs better on our training set.
How about its performance on the test set?

```
>>> dt.fit(X_train_fs, y_train)
>>> X_test_fs = fs.transform(X_test)
>>> y_pred_fs = dt.predict(X_test_fs)
>>> print "Accuracy:
    {:.3f}".format(metrics.accuracy_score(y_test,
```

```
y_pred_fs)) , "\n"  
Accuracy: 0.848
```

It seems that performance improvement on the training set did not hold for the evaluation set. This is always possible. In fact, performance could have decreased (recall overfitting). Our model is still the best. If we changed our model to use the one with the best performance in the testing set, we can never measure its performance, since the testing dataset could not be considered "unseen data" anymore.

Model selection

In the previous section we worked on ways to preprocess the data and select the most promising features. As we stated, selecting a good set of features is a crucial step to obtain good results. Now we will focus on another important step: selecting the algorithm parameters, known as **hyperparameters** to distinguish them from the parameters that are adjusted within the machine learning algorithm. Many machine learning algorithms include hyperparameters (from now on we will simply call them parameters) that guide certain aspects of the underlying method and have great impact on the results. In this section we will review some methods to help us obtain the best parameter configuration, a process known as model selection.

We will look back at the text-classification problem we addressed in [Chapter 2, Supervised Learning](#). In that example, we compounded a TF-IDF vectorizer alongside a multinomial **Naïve Bayes (NB)** algorithm to classify a set of newsgroup messages into a discrete number of categories. The `MultinomialNB` algorithm has one important parameter, named `alpha`, that adjusts the smoothing. We initially used the class with its default parameter values (`alpha = 1.0`) and obtained an accuracy

of 0.89. But when we set `alpha` to 0.01, we obtained a noticeable accuracy improvement to 0.92. Clearly, the configuration of the `alpha` parameter has great impact on the performance of the algorithm. How can we be sure 0.01 is the best value? Perhaps if we try other possible values, we could still obtain better results.

Let's start again with our text-classification problem, but for now we will only use a reduced number of instances. We will work only with 3,000 instances. We start by importing our `pylab` environment and loading the data.

```
>>> %pylab inline
>>> from sklearn.datasets import
fetch_20newsgroups
>>> news = fetch_20newsgroups(subset='all')
>>> n_samples = 3000
>>> X_train = news.data[:n_samples]
>>> y_train = news.target[:n_samples]
```

After that, we need to import the classes to construct our classifier.

```
>>> from sklearn.naive_bayes import
MultinomialNB
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.feature_extraction.text import
TfidfVectorizer
```

Then import the set of stop words and create a pipeline

that compounds the TF-IDF vectorizer and the Naïve Bayes algorithms (recall that we had a `stopwords_en.txt` file with a list of stop words).

```
>>> def get_stop_words():
>>>     result = set()
>>>     for line in open('stopwords_en.txt',
'r').readlines():
>>>         result.add(line.strip())
>>>     return result
>>> stop_words = get_stop_words()
>>> clf = Pipeline([('vect', TfidfVectorizer(
>>>                     stop_words=stop_words,
>>>                     token_pattern=r"\b[a-z0-9_\-\.]+\b[a-z][a-z0-9_\-\.]+\b",
>>>                     )),
>>> ('nb', MultinomialNB(alpha=0.01)),
>>> ])
```

If we evaluate our algorithm with a three-fold cross-validation, we obtain a mean score of around 0.811.

```
>>> from sklearn.cross_validation import
cross_val_score, KFold
>>> from scipy.stats import sem
>>> def evaluate_cross_validation(clf, X, y, K):
>>>     # create a k-fold cross validation
iterator of k=5 folds
>>>     cv = KFold(len(y), K, shuffle=True,
random_state=0)
>>>     # by default the score used is the one
returned by score
```

```
    method of the estimator (accuracy)
>>>     scores = cross_val_score(clf, X, y,
cv=cv)
>>>     print scores
>>>     print ("Mean score: {0:.3f} (+/-"
{1:.3f}).format(
>>>             np.mean(scores), sem(scores))
>>> evaluate_cross_validation(clf, X_train,
y_train, 3)
[ 0.814  0.815  0.804]
Mean score: 0.811 (+/-0.004)
```

It looks like we should train the algorithm with a list of different parameter values and keep the parameter value that achieves the best results. Let's implement a helper function to do that. This function will train the algorithm with a list of values, each time obtaining an accuracy score calculated by performing k-fold cross-validation on the training instances. After that, it will plot the training and testing scores as a function of the parameter values.

```
>>> def calc_params(X, y, clf, param_values,
param_name, K):
>>>     # initialize training and testing scores
with zeros
>>>     train_scores =
np.zeros(len(param_values))
>>>     test_scores =
np.zeros(len(param_values))
>>>
>>>     # iterate over the different parameter
values
```

```
>>>     for i, param_value in
enumerate(param_values):
>>>         print param_name, ' = ', param_value
>>>         # set classifier parameters
>>>         clf.set_params(**
{param_name:param_value})
>>>         # initialize the K scores obtained
for each fold
>>>             k_train_scores = np.zeros(K)
>>>             k_test_scores = np.zeros(K)
>>>             # create KFold cross validation
>>>             cv = KFold(n_samples, K,
shuffle=True, random_state=0)
>>>             # iterate over the K folds
>>>             for j, (train, test) in
enumerate(cv):
>>>                 clf.fit([X[k] for k in train],
y[train])
>>>                 k_train_scores[j] =
clf.score([X[k] for k in
train], y[train])
>>>                 k_test_scores[j] =
clf.score([X[k] for k in test],
y[test])
>>>                 train_scores[i] =
np.mean(k_train_scores)
>>>                 test_scores[i] =
np.mean(k_test_scores)
>>>
>>>             # plot the training and testing scores
in a log scale
>>>             plt.semilogx(param_values, train_scores,
alpha=0.4, lw=2,
c='b')
```

```

>>> plt.semilogx(param_values, test_scores,
alpha=0.4, lw=2,
                 c='g')
>>> plt.xlabel("Alpha values")
>>> plt.ylabel("Mean cross-validation
accuracy")
>>> # return the training and testing scores
on each parameter
value
>>> return train_scores, test_scores

```

The function accepts six arguments: the feature array, the target array, the classifier object to be used, the list of parameter values, the name of the parameter to adjust, and the number of K folds to be used in the crossvalidation evaluation.

Let's call this function; we will use numpy's `logspace` function to generate a list of alpha values spaced evenly on a log scale.

```

>>> alphas = np.logspace(-7, 0, 8)
>>> print alphas
[ 1.0000000e-07  1.0000000e-06
 1.0000000e-05  1.0000000e-04
 1.0000000e-03  1.0000000e-02   1.0000000e-01
 1.0000000e+00]

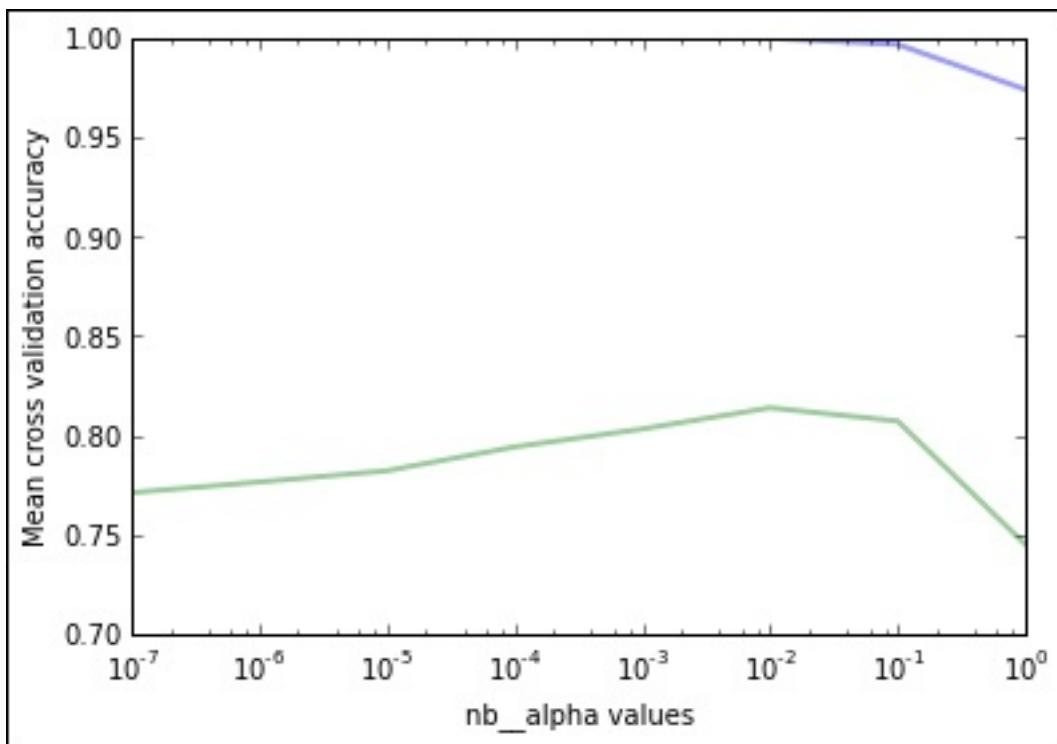
```

We will set the values of the `alpha` parameter of the NB classifier within the pipeline, which corresponds to the parameter name `nb_alpha`. We will use three folds for

the cross-validation.

```
>>> train_scores, test_scores =  
calc_params(X_train, y_train, clf, alphas,  
'nb_alpha', 3)
```

In the following figure, the line at the top corresponds to the training accuracy and the one at the bottom to the testing accuracy:



As expected, the training accuracy is always greater than the testing accuracy. We can see in the graph that the best testing accuracy is obtained with an alpha value in the

range of 10^{-2} and 10^{-1} . Below this range, the classifier shows signs of overfitting (the training accuracy is high but the testing accuracy is lower than it could be). Above this range, the classifier shows signs of underfitting (accuracy on the training set is lower than it could be).

It is worth mentioning that at this point a second pass could be performed in the range of 10^{-2} and 10^{-1} with a finer grid to find an ever better alpha value.

Let's print the scores vector to look at the actual values.

```
>>> print 'training scores: ', train_scores  
>>> print 'testing scores: ', test_scores  
training scores: [ 1. 1. 1. 1. 1. 0.99933333  
0.99633333 0.96933333]  
testing scores: [ 0.75 0.75666667 0.76433333  
0.77533333 0.78866667 0.811 0.81233333 0.753]
```

The best results are obtained with an `alpha` value of 0.1 (accuracy of 0.812).

We created a very useful function to graph and obtain the best parameter value for a classifier. Let's use it to adjust another classifier that uses a **Support Vector Machines (SVM)** instead of `MultinomialNB`:

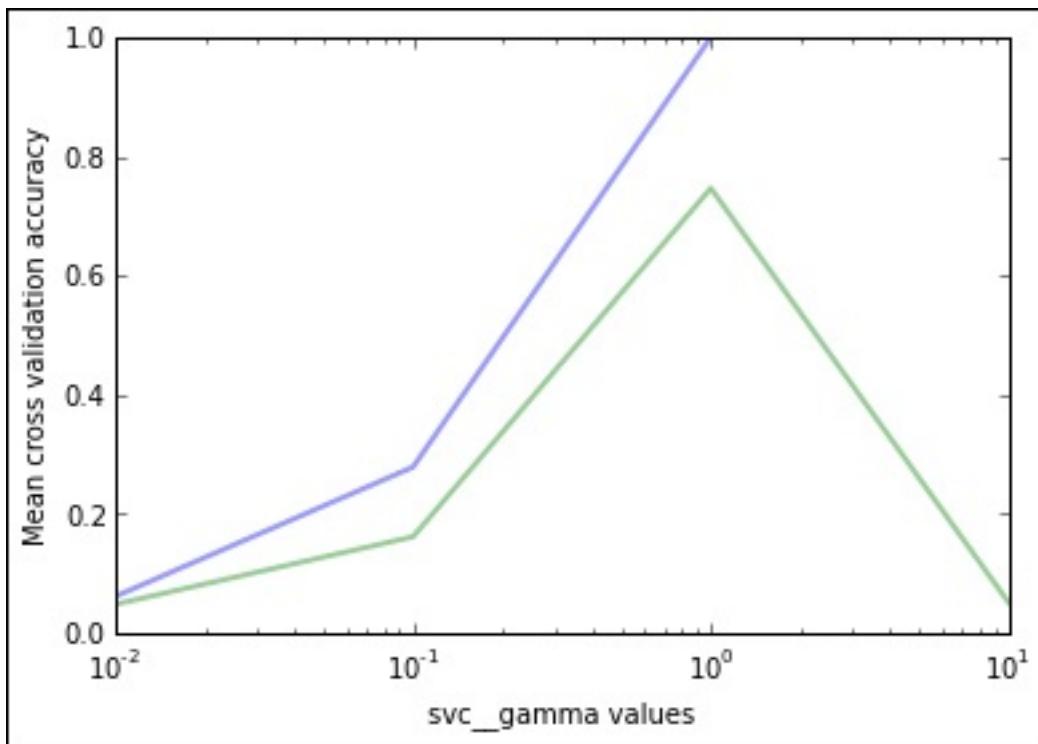
```
>>> from sklearn.svm import SVC  
>>>
```

```
>>> clf = Pipeline([
>>>     ('vect', TfidfVectorizer(
>>>         stop_words=stop_words,
>>>         token_pattern=ur"\b[a-zA-Z0-
9_\.]+\b",
>>>     )),
>>>     ('svc', SVC()),
>>> ])
```

We created a pipeline as before, but now we use the SVC classifier with its default values. Now we will use our calc_params function to adjust the gamma parameter.

```
>>> gammas = np.logspace(-2, 1, 4)
>>> train_scores, test_scores =
calc_params(X_train, y_train, clf,
gammas, 'svc_gamma', 3)
```

For gamma values lesser than one we have underfitting and for gamma values greater than one we have overfitting.



So the best result is for a `gamma` value of 1, where we obtain a training accuracy of 0.999 and a testing accuracy of 0.760.

If you take a closer look at the SVC class constructor parameters, we have other parameters, apart from gamma, that may also affect classifier performance. If we only adjust the gamma value, we implicitly state that the optimal `c` value is 1.0 (the default value that we did not explicitly set). Perhaps we could obtain better results with a new combination of `c` and `gamma` values. This opens a new degree of complexity; we should try all the parameter

combinations and keep the better one.

Grid search

To mitigate this problem, we have a very useful class named `GridSearchCV` within the `sklearn.grid_search` module. What we have been doing with our `calc_params` function is a kind of grid search in one dimension. With `GridSearchCV`, we can specify a grid of any number of parameters and parameter values to traverse. It will train the classifier for each combination and obtain a cross-validation accuracy to evaluate each one.

Let's use it to adjust the `C` and the `gamma` parameters at the same time.

```
>>> from sklearn.grid_search import GridSearchCV

>>> parameters = {
>>>     'svc__gamma': np.logspace(-2, 1, 4),
>>>     'svc__C': np.logspace(-1, 1, 3),
>>> }
>>> clf = Pipeline([
>>>     ('vect', TfidfVectorizer(
>>>             stop_words=stop_words,
>>>             token_pattern=ur"\b[a-z0-9_]-
>>> \.]+[a-z][a-z0-
>>>             9_-\.]+\b",
>>>         )),
>>>     ('svc', SVC()),
>>> ])
```

```
>>> gs = GridSearchCV(clf, parameters,
verbose=2, refit=False, cv=3)
```

Let's execute our grid search and print the best parameter values and scores.

```
>>> %time _ = gs.fit(X_train, y_train)
>>> gs.best_params_, gs.best_score_
CPU times: user 304.39 s, sys: 2.55 s, total:
306.94 s
Wall time: 306.56 s
({'svc__C': 10.0, 'svc__gamma':
0.1000000000000001}, 0.8116666666666665)
```

With the grid search, we obtained a better combination of `C` and `gamma` parameters, for values `10.0` and `0.10` respectively, with a three-fold cross-validation accuracy of `0.811`, which is much better than the best value we obtained (`0.76`) in the previous experiment by only adjusting `gamma` and keeping the `C` value at `1.0`.

At this point, we could continue performing experiments by trying not only to adjust other parameters of the SVC but also adjusting the parameters on `TfidfVectorizer`, which is also part of the estimator. Note that this additionally increases the complexity. As you might have noticed, the previous grid search experiment took about five minutes to finish. If we add new parameters to adjust, the time will increase exponentially. As a result, these

kinds of methods are very resource/time intensive; this is also the reason why we used only a subset of the total instances.

Parallel grid search

Grid search calculation grows exponentially with each parameter and its possible values we want to tune. We could reduce our response time if we calculate each of the combinations in parallel instead of sequentially, as we have done. In our previous example, we had four different values for `gamma` and three different values for `c`, summing up 12 parameter combinations. Additionally, we also needed to train each combination three times (in a three-fold cross-validation), so we summed up 36 trainings and evaluations. We could try to run these 36 tasks in parallel, since the tasks are independent.

Most modern computers have multiple cores that can be used to run tasks in parallel. We also have a very useful tool within IPython, called **IPython parallel**, that allows us to run independent tasks in parallel, each task in a different core of our machine. Let's do that with our text classifier example.

We will first declare a function that will persist all K folds for the cross-validation in different files. These files will be loaded by a process that will execute the corresponding fold. To do that, we will use the `joblib` library.

```
>>> from sklearn.externals import joblib
>>> from sklearn.cross_validation import
ShuffleSplit
>>> import os
>>> def persist_cv_splits(X, y, K=3,
name='data',
suffix=_cv_%03d.pkl"):
    """Dump K folds to filesystem."""
    cv_split_filenames = []
    # create KFold cross validation
    cv = KFold(n_samples, K, shuffle=True,
random_state=0)
    # iterate over the K folds
    for i, (train, test) in enumerate(cv):
        cv_fold = ([X[k] for k in train],
y[train], [X[k] for
                           k in test], y[test])
        cv_split_filename = name + suffix %
i
        cv_split_filename =
os.path.abspath(cv_split_filename)
        joblib.dump(cv_fold,
cv_split_filename)
    cv_split_filenames.append(cv_split_filename)
    return cv_split_filenames
cv_filenames = persist_cv_splits(X_train,
y_train, name='news')
```

The following function loads a particular fold and fits the classifier with the specified parameter set, returning the testing score. This function will be called by each of the parallel tasks.

```
>>> def compute_evaluation(cv_split_filename,
clf, params):
>>>
>>>     # All module imports should be executed
in the worker
        namespace
>>>     from sklearn.externals import joblib
>>>
>>>     # load the fold training and testing
partitions from the
        filesystem
>>>     X_train, y_train, X_test, y_test =
joblib.load(
        cv_split_filename, mmap_mode='c')
>>>
>>>     clf.set_params(**params)
>>>     clf.fit(X_train, y_train)
>>>     test_score = clf.score(X_test, y_test)
>>>     return test_score
```

Finally, the following function executes the grid search in parallel tasks. For each parameter combination (returned by the `IterGrid` iterator), it iterates over K folds and creates a task to compute the evaluation. It returns the parameter combinations alongside the tasks list.

```
>>> from sklearn.grid_search import IterGrid
>>>
>>> def parallel_grid_search(lb_view, clf,
cv_split_filenames, param_grid):
>>>     all_tasks = []
>>>     all_parameters =
list(IterGrid(param_grid))
>>>
>>>     # iterate over parameter combinations
>>>     for i, params in
enumerate(all_parameters):
>>>         task_for_params = []
>>>         # iterate over the K folds
>>>         for j, cv_split_filename in
enumerate(cv_split_filenames):
>>>             t = lb_view.apply(
>>>                 compute_evaluation,
cv_split_filename, clf,
params)
>>>             task_for_params.append(t)
>>>
>>>         all_tasks.append(task_for_params)
>>>
>>>     return all_parameters, all_tasks
```

Now we use IPython parallel to get the client and a load balanced view. We must first create a local cluster of N engines (one for each core of your machine) using the cluster tab in the IPython Notebook. Then we create the client and the view and execute our parallel_grid_search function.

```
>>> from sklearn.svm import SVC
>>> from IPython.parallel import Client
>>>
>>> client = Client()
>>> lb_view = client.load_balanced_view()
>>>
>>> all_parameters, all_tasks =
parallel_grid_search(
    lb_view, clf, cv_filenames, parameters)
```

IPython parallel will start to run the tasks in parallel. We can use this to monitor the progress of the whole task group.

```
>>> def print_progress(tasks):
>>>     progress = np.mean([task.ready() for
task_group in tasks
for task in
task_group])
>>>     print "Tasks completed: {}%".format(100
* progress)
```

After all the tasks are completed, use the following function:

```
>>> print_progress(all_tasks)
Tasks completed: 100.0%
```

We can define a function that computes the mean score of the completed tasks.

```
>>> def find_bests(all_parameters, all_tasks,
```

```

n_top=5) :
>>>     """Compute the mean score of the
completed tasks"""
>>>     mean_scores = []
>>>
>>>     for param, task_group in
zip(all_parameters, all_tasks):
>>>         scores = [t.get() for t in
task_group if t.ready()]
>>>         if len(scores) == 0:
>>>             continue
>>>         mean_scores.append((np.mean(scores),
param))
>>>
>>>     return sorted(mean_scores, reverse=True)
[:n_top]
>>> print find_bests(all_parameters, all_tasks)

[(0.8173333333333336, {'svc_gamma':
0.1000000000000001, 'svc_C': 10.0}),
(0.7873333333333333, {'svc_gamma': 1.0,
'svc_C': 10.0}), (0.7600000000000012,
{'svc_gamma': 1.0, 'svc_C': 1.0}),
(0.3009999999999999, {'svc_gamma': 0.01,
'svc_C': 10.0}), (0.1993333333333333,
{'svc_gamma': 0.1000000000000001, 'svc_C':
1.0})]

```

You can observe that we computed the same results as in the previous section, but in half the time (if you used two cores) or in a quarter of the time (if you used four cores).

Summary

In this chapter we reviewed two important methods to improve our results when applying machine learning algorithms: feature selection and model selection. First, we used different techniques to preprocess data, extract features, and select the most promising features. Then we used techniques to automatically calculate the most promising hyperparameters of machine learning algorithms and used methods to parallelize these calculations.

The reader must be aware that this book covered only the main machine learning lines and some of their methods. Keep in mind that there is much more than supervised and unsupervised learning. For example:

- Semi-supervised learning methods are the middle ground between supervised and unsupervised learning. They combine small amounts of annotated data with huge amounts of unlabeled data. Usually, unlabeled data can reveal the underlying distribution of elements and obtain better results in combination with a small, labeled dataset.
- Active learning is a particular case within semi-supervised methods. Again, it is useful when labeled

data is scarce or hard to obtain. In active learning, the algorithm actively queries a human expert to answer the label of certain unlabeled instances, and thus learn the concept over a reduced set of labeled instances.

- Reinforcement learning proposes methods where an agent learns from feedback (rewards or reinforcements) after performing actions within an environment. The agent learns to perform a task by trying to maximize the cumulative reward. These methods have been very successful in robotics and video games.
- Sequential classification (very commonly used in **Natural Language Processing (NLP)**) assigns a sequence of labels to a sequence of items; for example, the parts of speech of the words in a sentence.

Besides these, there are lots of supervised learning methods with radically different approaches to those we presented; for example, neural networks, maximum entropy models, memory-based models, and rule-based models. Machine learning is a very active research area with a growing literature; there are many books and courses that the reader can use to go deeper into the theory and details.

Scikit-learn has many of these algorithms implemented, and lacks others, but expect its active and enthusiastic contributors to build them soon. We encourage the reader to be part of the community!

Part 2. Module 2

scikit-learn Cookbook

*Over 50 recipes to incorporate scikit-learn into
every step of the data science pipeline, from feature
extraction to model building and model evaluation*

Chapter 1. Premodel Workflow

This chapter will cover the following topics:

- Getting sample data from external sources
- Creating sample data for toy analysis
- Confirming the characteristics of created data
- Scaling data to the standard normal
- Creating binary features through thresholding
- Working with categorical variables
- Binarizing label features
- Imputing missing values through various strategies
- Using Pipelines for multiple preprocessing steps
- Reducing dimensionality with PCA
- Using factor analytics for decomposition
- Kernel PCA for nonlinear dimensionality reduction
- Using truncated SVD to reduce dimensionality
- Decomposition to classify with DictionaryLearning
- Putting it all together with Pipelines
- Using Gaussian processes for regression
- Defining the Gaussian process object directly
- Using stochastic gradient descent for regression

Introduction

This chapter discusses setting data, preparing data, and premodel dimensionality reduction. These are not the attractive parts of **machine learning (ML)**, but they often turn out to be what determines if a model will work or not.

There are three main parts to the chapter. Firstly, we'll create fake data; this might seem trivial, but creating fake data and fitting models to fake data is an important step in model testing. It's more useful in situations where we implement an algorithm from scratch, but I'll cover it here for completeness, and in the event you don't have data of your own, you can just create it. Secondly, we'll look at broadly handling data transformations as a preprocessing step, which includes data imputation, categorical variable encoding, and so on. Thirdly, we'll look at situations where we have a large number of features relative to the number of observations we have.

This chapter, especially the first half, will set the stage for the later chapters. In order to use scikit-learn, data is required. The first two sections will discuss acquiring the data; the rest of the first half will discuss preparing this data for use.

Tip

This book is written using scikit-learn 0.15, NumPy 1.9, and pandas 0.13. There are other packages used as well, so it's advisable that you refer to the installation instructions included in this book.

Getting sample data from external sources

If possible, try working with a familiar dataset while working through this book; in order to level the field, built-in datasets will be used. The built-in datasets can be used as stand-ins to test several different modeling techniques such as regression and classification. These are, for the most part, famous datasets. This is very useful as papers in various fields will often use these datasets for authors to put forth how their model fits as compared to other models.

Tip

I recommend you use IPython to run these commands as they are presented. Muscle memory is important, and it's best to get to the point where basic commands take no extra mental effort. An even better way might be to run IPython Notebook. If you do, make sure to use the `%matplotlib inline` command; this will allow you to see the plots in Notebook.

Getting ready

The datasets in scikit-learn are contained within the `datasets` module. Use the following command to import these datasets:

```
>>> from sklearn import datasets  
>>> import numpy as np
```

From within IPython, run `datasets.*?`, which will list everything available within the `datasets` module.

How to do it...

There are two main types of data within the `datasets` module. Smaller test datasets are included in the `sklearn` package and can be viewed by running `datasets.load_*`? Larger datasets are also available for download as required. The latter are not included in `sklearn` by default; however, at times, they are better to test models and algorithms due to sufficient complexity to represent realistic situations.

Datasets are included with `sklearn` by default; to view these datasets, run `datasets.load_*`? There are other types of datasets that must be fetched. These datasets are larger, and therefore, they do not come within the package. This said, they are often better to test algorithms that might be used in the wild.

First, load the `boston` dataset and examine it:

```
>>> boston = datasets.load_boston()  
>>> print boston.DESCR #output omitted due to length
```

`DESCR` will present a basic overview of the data to give you some context.

Next, fetch a dataset:

```
>>> housing =
datasets.fetch_california_housing()
downloading Cal. housing from
http://lib.stat.cmu.edu [...]

>>> print housing.DESCR #output omitted due to
length
```

How it works...

When these datasets are loaded, they aren't loaded as NumPy arrays. They are of type `Bunch`. A **Bunch** is a common data structure in Python. It's essentially a dictionary with the keys added to the object as attributes.

To access the data using the (surprise!) `data` attribute, which is a NumPy array containing the independent variables, the `target` attribute has the dependent variable:

```
>>> x, y = boston.data, boston.target
```

There are various implementations available on the Web for the `Bunch` object; it's not too difficult to write on your own. scikit-learn defines `Bunch` (as of this writing) in the base module.

It's available in GitHub at <https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/datasets/base.py>.

There's more...

When you fetch a dataset from an external source it will, by default, place the data in your home directory under `scikit_learn_data/`; this behavior is configurable in two ways:

- To modify the default behavior, set the `SCIKIT_LEARN_DATA` environment variable to point to the desired folder.
- The first argument of the fetch methods is `data_home`, which will specify the home folder on a case-by-case basis.

It is easy to check the default location by calling `datasets.get_data_home()`.

See also

The **UCI Machine Learning Repository** is a great place to find sample datasets. Many of the datasets in scikit-learn are hosted here; however, there are more datasets available. Other notable sources include KDD, your local government agency, and Kaggle competitions.

Creating sample data for toy analysis

I will again implore you to use some of your own data for this book, but in the event you cannot, we'll learn how we can use scikit-learn to create toy data.

Getting ready

Very similar to getting built-in datasets, fetching new datasets, and creating sample datasets, the functions that are used follow the naming convention `make_<the data set>`. Just to be clear, this data is purely artificial:

```
>>> datasets.make_*
datasets.make_biclusters
datasets.make_blobs
datasets.make_checkerboard
datasets.make_circles
datasets.make_classification
...
...
```

To save typing, import the `datasets` module as `d`, and `numpy` as `np`:

```
>>> import sklearn.datasets as d
>>> import numpy as np
```

How to do it...

This section will walk you through the creation of several datasets; the following *How it works...* section will confirm the purported characteristics of the datasets. In addition to the sample datasets, these will be used throughout the book to create data with the necessary characteristics for the algorithms on display.

First, the stalwart—regression:

```
>>> reg_data = d.make_regression()
```

By default, this will generate a tuple with a 100 x 100 matrix – 100 samples by 100 features. However, by default, only 10 features are responsible for the target data generation. The second member of the tuple is the target variable.

It is also possible to get more involved. For example, to generate a 1000 x 10 matrix with five features responsible for the target creation, an underlying bias factor of 1.0, and 2 targets, the following command will be run:

```
>>> complex_reg_data = d.make_regression(1000,  
10, 5, 2, 1.0)  
>>> complex_reg_data[0].shape  
(1000, 10)
```

Classification datasets are also very simple to create. It's simple to create a base classification set, but the basic case is rarely experienced in practice—most users don't convert, most transactions aren't fraudulent, and so on. Therefore, it's useful to explore classification on unbalanced datasets:

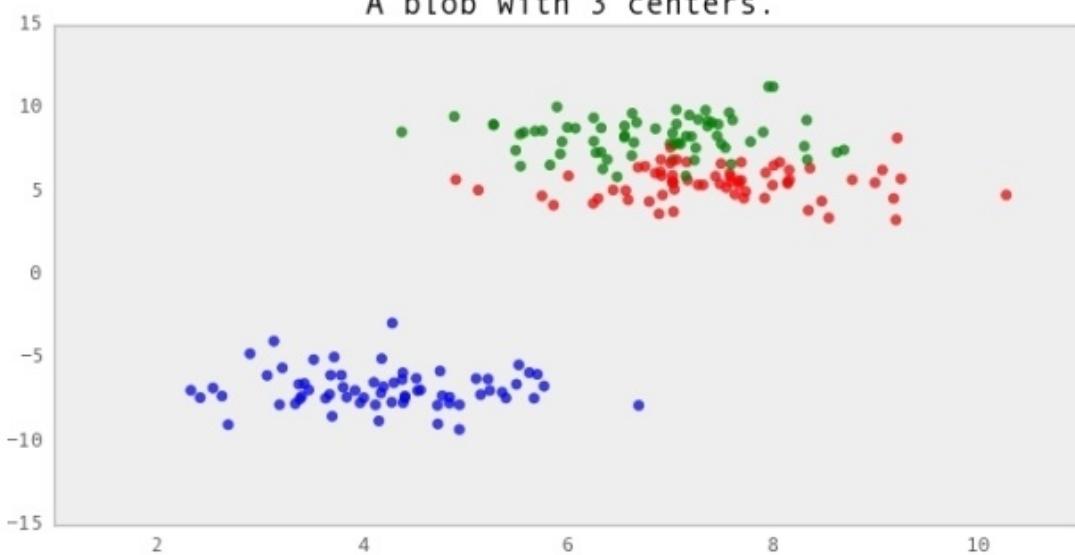
```
>>> classification_set =
d.make_classification(weights=[0.1])
>>> np.bincount(classification_set[1])
array([10, 90])
```

Clusters will also be covered. There are actually several functions to create datasets that can be modeled by different cluster algorithms. For example, `blobs` are very easy to create and can be modeled by K-Means:

```
>>> blobs = d.make_blobs()
```

This will look like the following:

A blob with 3 centers.



How it works...

Let's walk you through how scikit-learn produces the regression dataset by taking a look at the source code (with some modifications for clarity). Any undefined variables are assumed to have the default value of `make_regression`.

It's actually surprisingly simple to follow.

First, a random array is generated with the size specified when the function is called:

```
>>> x = np.random.randn(n_samples, n_features)
```

Given the basic dataset, the target dataset is then generated:

```
>>> ground_truth = np.zeros((n_features, n_target))
>>> ground_truth[:n_informative, :] =
100*np.random.rand(n_informative, n_target)
```

The dot product of `x` and `ground_truth` are taken to get the final target values. Bias, if any, is added at this time:

```
>>> y = np.dot(x, ground_truth) + bias
```

Note

The dot product is simply a matrix multiplication. So, our final dataset will have `n_samples`, which is the number of rows from the dataset, and `n_target`, which is the number of target variables.

Due to NumPy's broadcasting, bias can be a scalar value, and this value will be added to every sample.

Finally, it's a simple matter of adding any noise and shuffling the dataset. Voilà, we have a dataset perfect to test regression.

Scaling data to the standard normal

A preprocessing step that is almost recommended is to scale columns to the standard normal. The **standard normal** is probably the most important distribution of all statistics.

If you've ever been introduced to statistics, you must have almost certainly seen **z-scores**. In truth, that's all this recipe is about—transforming our features from their endowed distribution into z-scores.

Getting ready

The act of scaling data is extremely useful. There are a lot of machine learning algorithms, which perform differently (and incorrectly) in the event the features exist at different scales. For example, SVMs perform poorly if the data isn't scaled because it uses a distance function in its optimization, which is biased if one feature varies from 0 to 10,000 and the other varies from 0 to 1.

The preprocessing module contains several useful functions to scale features:

```
>>> from sklearn import preprocessing  
>>> import numpy as np # we'll need it later
```

How to do it...

Continuing with the `boston` dataset, run the following commands:

```
>>> x[:, :3].mean(axis=0) #mean of the first 3  
features  
array([ 3.59376071, 11.36363636,  
11.13677866])  
>>> x[:, :3].std(axis=0)  
array([ 8.58828355, 23.29939569,  
6.85357058])
```

There's actually a lot to learn from this initially. Firstly, the first feature has the smallest mean but varies even more than the third feature. The second feature has the largest mean and standard deviation—it takes the widest spread of values:

```
>>> x_2 = preprocessing.scale(x[:, :3])  
  
>>> x_2.mean(axis=0)  
array([ 6.34099712e-17, -6.34319123e-16,  
-2.68291099e-15])  
  
>>> x_2.std(axis=0)  
array([ 1., 1., 1.])
```

How it works...

The center and scaling function is extremely simple. It merely subtracts the mean and divides by the standard deviation:

$$x = \frac{x - \bar{x}}{\sigma}$$

In addition to a function, there is also a center and scaling class that is easy to invoke, and this is particularly useful when used in conjunction with the Pipelines mentioned later. It's also useful for the center and scaling class to persist across individual scaling:

```
>>> my_scaler = preprocessing.StandardScaler()
>>> my_scaler.fit(X[:, :3])
>>> my_scaler.transform(X[:, :3]).mean(axis=0)
array([-6.34099712e-17, -6.34319123e-16,
-2.68291099e-15])
```

Scaling features to mean 0 , and standard deviation 1 isn't the only useful type of scaling. Preprocessing also contains a `MinMaxScaler` class, which will scale the data within a certain range:

```
>>> my_minmax_scaler =  
preprocessing.MinMaxScaler()  
>>> my_minmax_scaler.fit(X[:, :3])  
>>> my_minmax_scaler.transform(X[:,  
:3]).max(axis=0)  
array([ 1.,  1.,  1.])
```

It's very simple to change the minimum and maximum values of the `MinMaxScaler` class from its default of 0 and 1, respectively:

```
>>> my_odd_scaler =  
preprocessing.MinMaxScaler(feature_range=(-3.14,  
3.14))
```

Furthermore, another option is **normalization**. This will scale each sample to have a length of 1. This is different from the other types of scaling done previously, where the features were scaled. Normalization is illustrated in the following command:

```
>>> normalized_x = preprocessing.normalize(X[:,  
:3])
```

If it's not apparent why this is useful, consider the Euclidian distance (a measure of similarity) between three of the samples, where one sample has the values (1, 1, 0), another has (3, 3, 0), and the final has (1, -1, 0).

The distance between the 1st and 3rd vector is less than the distance between the 1st and 2nd though the 1st and 3rd are orthogonal, whereas the 1st and 2nd only differ by a scalar factor of 3. Since distances are often used as measures of similarity, not normalizing the data first will be misleading..

There's more...

Imputation is a very deep subject. Here are a few things to consider when using scikit-learn's implementation.

Creating idempotent scalar objects

It is possible to scale the mean and/or variance in the `StandardScaler` instance. For instance, it's possible (though not useful) to create a `StandardScaler` instance, which simply performs the identity transformation:

```
>>> my_useless_scaler =
preprocessing.StandardScaler(with_mean=False,
with_std=False)
>>> transformed_sd = my_useless_scaler
                    .fit_transform(X[:, :3]).std(axis=0)
>>> original_sd = X[:, :3].std(axis=0)
>>> np.array_equal(transformed_sd, original_sd)
```

Handling sparse imputations

Sparse matrices aren't handled differently from normal matrices when doing scaling. This is because to mean center the data, the data will have its 0s altered to nonzero values, thus the matrix will no longer be sparse:

```
>>> matrix = scipy.sparse.eye(1000)
```

```
>>> preprocessing.scale(matrix)
...
ValueError: Cannot center sparse matrices: pass
'with_mean=False' instead See docstring for
motivation and alternatives.
```

As noted in the error, it is possible to scale a sparse matrix with_std only:

```
>>> preprocessing.scale(matrix, with_mean=False)
<1000x1000 sparse matrix of type '<type
'numpy.float64'>'
      with 1000 stored elements in Compressed
Sparse Row format>
```

The other option is to call `todense()` on the array. However, this is dangerous because the matrix is already sparse for a reason, and it will potentially cause a memory error.

Creating binary features through thresholding

In the last recipe, we looked at transforming our data into the standard normal distribution. Now, we'll talk about another transformation, one that is quite different.

Instead of working with the distribution to standardize it, we'll purposely throw away data; but, if we have good reason, this can be a very smart move. Often, in what is ostensibly continuous data, there are discontinuities that can be determined via binary features.

Getting ready

Creating binary features and outcomes is a very useful method, but it should be used with caution. Let's use the `boston` dataset to learn how to create turn values in binary outcomes.

First, load the `boston` dataset:

```
>>> from sklearn import datasets  
>>> boston = datasets.load_boston()  
>>> import numpy as np
```

How to do it...

Similar to scaling, there are two ways to binarize features in scikit-learn:

- `preprocessing.binarize` # (a function)
- `preprocessing.Binarizer` # (a class)

The `boston` dataset's target variable is the median value of houses in thousands. This dataset is good to test regression and other continuous predictors, but consider a situation where we want to simply predict if a house's value is more than the overall mean. To do this, we will want to create a threshold value of the mean. If the value is greater than the mean, produce a 1; if it is less, produce a 0:

```
>>> from sklearn import preprocessing
>>> new_target =
preprocessing.binarize(boston.target,
threshold=boston.target.mean())
>>> new_target[:5]
array([ 1.,  0.,  1.,  1.,  1.])
```

This was easy, but let's check to make sure it worked correctly:

```
>>> (boston.target[:5] >
```

```
boston.target.mean()).astype(int)
array([1, 0, 1, 1, 1])
```

Given the simplicity of the operation in NumPy, it's a fair question to ask why you will want to use the built-in functionality of scikit-learn. Pipelines, covered in the *Using Pipelines for multiple preprocessing steps* recipe, will go far to explain this; in anticipation of this, let's use the `Binarizer` class:

```
>>> bin =
preprocessing.Binarizer(boston.target.mean())
>>> new_target =
bin.fit_transform(boston.target)
>>> new_target[:5]
array([ 1.,  0.,  1.,  1.,  1.])
```

How it works...

Hopefully, this is pretty obvious; but under the hood, scikit-learn creates a conditional mask that is `True` if the value in the array in question is more than the threshold. It then updates the array to 1 where the condition is met, and 0 where it is not.

There's more...

Let's also learn about sparse matrices and the `fit` method.

Sparse matrices

Sparse matrices are special in that zeros aren't stored; this is done in an effort to save space in memory. This creates an issue for the binarizer, so to combat it, a special condition for the binarizer for sparse matrices is that the threshold cannot be less than zero:

```
>>> from scipy.sparse import coo
>>> spar = coo.coo_matrix(np.random.binomial(1,
... .25, 100))
>>> preprocessing.binarize(spar, threshold=-1)
ValueError: Cannot binarize a sparse matrix with
threshold < 0
```

The fit method

The `fit` method exists for the binarizer transformation, but it will not fit anything, it will simply return the object.

Working with categorical variables

Categorical variables are a problem. On one hand they provide valuable information; on the other hand, it's probably text—either the actual text or integers corresponding to the text—like an index in a lookup table.

So, we clearly need to represent our text as integers for the model's sake, but we can't just use the `id` field or naively represent them. This is because we need to avoid a similar problem to the *Creating binary features through thresholding* recipe. If we treat data that is continuous, it must be interpreted as continuous.

Getting ready

The `boston` dataset won't be useful for this section. While it's useful for feature binarization, it won't suffice for creating features from categorical variables. For this, the `iris` dataset will suffice.

For this to work, the problem needs to be turned on its head. Imagine a problem where the goal is to predict the sepal width; therefore, the species of the flower will probably be useful as a feature.

Let's get the data sorted first:

```
>>> from sklearn import datasets  
>>> iris = datasets.load_iris()  
>>> X = iris.data  
>>> y = iris.target
```

Now, with X and Y being as they normally will be, we'll operate on the data as one:

```
>>> import numpy as np  
>>> d = np.column_stack((X, y))
```

How to do it...

Convert the text columns to three features:

```
>>> from sklearn import preprocessing
>>> text_encoder = preprocessing.OneHotEncoder()
>>> text_encoder.fit_transform(d[:, -1:]).toarray()[:5]
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

How it works...

The encoder creates additional features for each categorical variable, and the value returned is a sparse matrix. The result is a sparse matrix by definition; each row of the new features has 0 everywhere, except for the column whose value is associated with the feature's category. Therefore, it makes sense to store this data in a sparse matrix.

`text_encoder` is now a standard scikit-learn model, which means that it can be used again:

```
>>> text_encoder.transform(np.ones((3, 1))).toarray()
array([[ 0.,  1.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  1.,  0.]])
```

There's more...

Other options exist to create categorical variables in scikit-learn and Python at large. `DictVectorizer` is a good option if you like to limit the dependencies of your projects to only scikit-learn and you have a fairly simple encoding scheme. However, if you require more sophisticated categorical encoding, `patsy` is a very good option.

DictVectorizer

Another option is to use `DictVectorizer`. This can be used to directly convert strings to features:

```
>>> from sklearn.feature_extraction import  
DictVectorizer  
>>> dv = DictVectorizer()  
>>> my_dict = [ {'species': iris.target_names[i]}  
for i in y]  
>>> dv.fit_transform(my_dict).toarray() [:5]  
array([[ 1.,  0.,  0.],  
       [ 1.,  0.,  0.],  
       [ 1.,  0.,  0.],  
       [ 1.,  0.,  0.],  
       [ 1.,  0.,  0.]])
```

Tip

Dictionaries can be viewed as a sparse matrix. They only

contain entries for the nonzero values.

Patsy

patsy is another package useful to encode categorical variables. Often used in conjunction with StatsModels, patsy can turn an array of strings into a design matrix.

Tip

This section does not directly pertain to scikit-learn; therefore, skipping it is okay without impacting the understanding of how scikit-learn works.

For example, `dm = patsy.design_matrix("x + y")` will create the appropriate columns if `x` or `y` are strings. If they aren't, `C(x)` inside the formula will signify that it is a categorical variable.

For example, `iris.target` can be interpreted as a continuous variable if we don't know better. Therefore, use the following command:

```
>>> import patsy
>>> patsy.dmatrix("0 + C(species)", {'species': iris.target})
DesignMatrix with shape (150, 3)
C(species) [0]  C(species) [1]  C(species) [2]
      1              0              0
      1              0              0
```

1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0

[. . .]

Binarizing label features

In this recipe, we'll look at working with categorical variables in a different way. In the event that only one or two categories of the feature are important, it might be wise to avoid the extra dimensionality, which might be created if there are several categories.

Getting ready

There's another way to work with categorical variables. Instead of dealing with the categorical variables using `OneHotEncoder`, we can use `LabelBinarizer`. This is a combination of thresholding and working with categorical variables.

To show how this works, load the `iris` dataset:

```
>>> from sklearn import datasets as d
>>> iris = d.load_iris()
>>> target = iris.target
```

How to do it...

Import the `LabelBinarizer()` method and create an object:

```
>>> from sklearn.preprocessing import  
LabelBinarizer  
>>> label_binarizer = LabelBinarizer()
```

Now, simply transform the target outcomes to the new feature space:

```
>>> new_target =  
label_binarizer.fit_transform(target)
```

Let's look at `new_target` and the `label_binarizer` object to get a feel of what happened:

```
>>> new_target.shape  
(150, 3)  
>>> new_target[:5]  
array([[1, 0, 0],  
       [1, 0, 0],  
       [1, 0, 0],  
       [1, 0, 0],  
       [1, 0, 0]])  
  
>>> new_target[-5:]  
array([[0, 0, 1],  
       [0, 0, 1],  
       [0, 0, 1],
```

```
[0, 0, 1],  
[0, 0, 1]])  
  
>>> label_binarizer.classes_  
array([0, 1, 2])
```

How it works...

The `iris` target has a cardinality of 3, that is, it has three unique values. When `LabelBinarizer` converts the vector $N \times 1$ into the vector $N \times C$, where C is the cardinality of the $N \times 1$ dataset, it is important to note that once the object has been fit, introducing unseen values in the transformation will throw an error:

```
>>> label_binarizer.transform([4])
[...]
ValueError: classes [0 1 2] mismatch with the
labels [4] found in the data
```

There's more...

Zero and one do not have to represent the positive and negative instances of the target value. For example, if we want positive values to be represented by 1,000, and negative values to be represented by -1,000, we'd simply make the designation when we create `label_binarizer`:

```
>>> label_binarizer =  
LabelBinarizer(neg_label=-1000, pos_label=1000)  
>>> label_binarizer.fit_transform(target)[:5]  
array([[ 1000, -1000, -1000],  
       [ 1000, -1000, -1000],  
       [ 1000, -1000, -1000],  
       [ 1000, -1000, -1000],  
       [ 1000, -1000, -1000]])
```

Tip

The only restriction on the positive and negative values is that they must be integers.

Imputing missing values through various strategies

Data imputation is critical in practice, and thankfully there are many ways to deal with it. In this recipe, we'll look at a few of the strategies. However, be aware that there might be other approaches that fit your situation better.

This means scikit-learn comes with the ability to perform fairly common imputations; it will simply apply some transformations to the existing data and fill the NAs. However, if the dataset is missing data, and there's a known reason for this missing data—for example, response times for a server that times out after 100ms—it might be better to take a statistical approach through other packages such as the Bayesian treatment via PyMC, the Hazard Models via Lifelines, or something home-grown.

Getting ready

The first thing to do to learn how to input missing values is to create missing values. NumPy's masking will make this extremely simple:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> iris_X = iris.data
>>> masking_array = np.random.binomial(1, .25,
                                      iris_X.shape).astype(bool)
>>> iris_X[masking_array] = np.nan
```

To unravel this a bit, in case NumPy isn't too familiar, it's possible to index arrays with other arrays in NumPy. So, to create the random missing data, a random Boolean array is created, which is of the same shape as the `iris` dataset. Then, it's possible to make an assignment via the masked array. It's important to note that because a random array is used, it is likely your `masking_array` will be different from what's used here.

To make sure this works, use the following command (since we're using a random mask, it might not match directly):

```
>>> masking_array[:5]
array([[False, False, False, False],
```

```
[False, False, False, False],
[False, False, False, False],
[ True, False, False, False],
[False, False, False, False]],
dtype=bool)
>>> iris_X [:5]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ nan,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2]])
```

How to do it...

A theme prevalent throughout this book (due to the theme throughout scikit-learn) is reusable classes that fit and transform datasets and that can subsequently be used to transform unseen datasets. This is illustrated as follows:

```
>>> from sklearn import preprocessing
>>> impute = preprocessing.Imputer()
>>> iris_X_prime = impute.fit_transform(iris_X)
>>> iris_X_prime[:5]
array([[ 5.1      ,  3.5      ,  1.4      ,
0.2      ],
       [ 4.9      ,  3.        ,  1.4      ,
0.2      ],
       [ 4.7      ,  3.2      ,  1.3      ,
0.2      ],
       [ 5.87923077,  3.1      ,  1.5      ,
0.2      ],
       [ 5.        ,  3.6      ,  1.4      ,
0.2      ]])
```

Notice the difference in the position [3, 0]:

```
>>> iris_X_prime[3, 0]
5.87923077
>>> iris_X[3, 0]
nan
```

How it works...

The imputation works by employing different strategies. The default is `mean`, but in total there are:

- `mean` (default)
- `median`
- `most_frequent` (the mode)

scikit-learn will use the selected strategy to calculate the value for each non-missing value in the dataset. It will then simply fill the missing values.

For example, to redo the `iris` example with the `median` strategy, simply reinitialize `impute` with the new strategy:

```
>>> impute =
preprocessing.Imputer(strategy='median')
>>> iris_X_prime = impute.fit_transform(iris_X)
>>> iris_X_prime[:5]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 5.8,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2]])
```

If the data is missing values, it might be inherently dirty in other places. For instance, in the example in the preceding *How to do it...* section, `np.nan` (the default

missing value) was used as the missing value, but missing values can be represented in many ways. Consider a situation where missing values are `-1`. In addition to the strategy to compute the missing value, it's also possible to specify the missing value for the imputer. The default is `NaN`, which will handle `np.nan` values.

To see an example of this, modify `iris_X` to have `-1` as the missing value. It sounds crazy, but since the `iris` dataset contains measurements that are always possible, many people will fill the missing values with `-1` to signify they're not there:

```
>>> iris_X[np.isnan(iris_X)] = -1
>>> iris_X[:5]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [-1. ,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2]])
```

Filling these in is as simple as the following:

```
>>> impute =
preprocessing.Imputer(missing_values=-1)
>>> iris_X_prime = impute.fit_transform(iris_X)
>>> iris_X_prime[:5]
array([[ 5.1          ,  3.5          ,  1.4          ,
        0.2          ],
       [ 4.9          ,  3.           ,  1.4          ,
        0.2          ],
       [ 4.7          ,  3.2          ,  1.3          ,
        0.2          ],
       [-1.           ,  3.1          ,  1.5          ,
        0.2          ],
       [ 5.           ,  3.6          ,  1.4          ,
        0.2          ]])
```

```
0.2      ] ,  
        [ 4.7      ,  3.2      ,  1.3      ,  
0.2      ] ,  
        [ 5.87923077,  3.1      ,  1.5      ,  
0.2      ] ,  
        [ 5.      ,  3.6      ,  1.4      ,  
0.2      ]) )
```

There's more...

pandas also provides a functionality to fill missing data. It actually might be a bit more flexible, but it is less reusable:

```
>>> import pandas as pd
>>> iris_X[masking_array] = np.nan
>>> iris_df = pd.DataFrame(iris_X,
columns=iris.feature_names)
>>> iris_df.fillna(iris_df.mean())['sepal length
(cm)'].head(5)
0    5.100000
1    4.900000
2    4.700000
3    5.879231
4    5.000000
Name: sepal length (cm), dtype: float64
```

To mention its flexibility, `fillna` can be passed any sort of statistic, that is, the strategy is more arbitrarily defined:

```
>>> iris_df.fillna(iris_df.max())['sepal length
(cm)'].head(5)
0    5.1
1    4.9
2    4.7
3    7.9
4    5.0
Name: sepal length (cm), dtype: float64
```

Using Pipelines for multiple preprocessing steps

Pipelines are (at least to me) something I don't think about using often, but are useful. They can be used to tie together many steps into one object. This allows for easier tuning and better access to the configuration of the entire model, not just one of the steps.

Getting ready

This is the first section where we'll combine multiple data processing steps into a single step. In scikit-learn, this is known as a Pipeline. In this section, we'll first deal with missing data via imputation; however, after that, we'll scale the data to get a mean of zero and a standard deviation of one.

Let's create a dataset that is missing some values, and then we'll look at how to create a Pipeline:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> mat = datasets.make_spd_matrix(10)
>>> masking_array = np.random.binomial(1, .1,
mat.shape).astype(bool)
>>> mat[masking_array] = np.nan
>>> mat[:4, :4]
array([[ 0.56716186, -0.20344151,          nan,
-0.22579163],
       [          nan,   1.98881836, -2.25445983,
1.27024191],
       [ 0.29327486, -2.25445983,   3.15525425,
-1.64685403],
       [-0.22579163,   1.27024191, -1.64685403,
1.32240835]])
```

Great, now we can create a Pipeline.

How to do it...

Without Pipelines, the process will look something like the following:

```
>>> from sklearn import preprocessing
>>> impute = preprocessing.Imputer()
>>> scaler = preprocessing.StandardScaler()
>>> mat_imputed = impute.fit_transform(mat)
>>> mat_imputed[:4, :4]
array([[ 0.56716186, -0.20344151, -0.80554023,
-0.22579163],
       [ 0.04235695,  1.98881836, -2.25445983,
1.27024191],
       [ 0.29327486, -2.25445983,  3.15525425,
-1.64685403],
       [-0.22579163,  1.27024191, -1.64685403,
1.32240835]])
>>> mat_imp_and_scaled =
scaler.fit_transform(mat_imputed)
array([[ 2.235e+00,  -6.291e-01,    1.427e-16,
-7.496e-01],
       [ 0.000e+00,   1.158e+00,   -9.309e-01,
9.072e-01],
       [ 1.068e+00,  -2.301e+00,    2.545e+00,
-2.323e+00],
       [ -1.142e+00,   5.721e-01,   -5.405e-01,
9.650e-01]])
```

Notice that the prior missing value is 0. This is expected because this value was imputed using the mean strategy,

and scaling subtracts the mean.

Now that we've looked at a non-Pipeline example, let's look at how we can incorporate a Pipeline:

```
>>> from sklearn import pipeline  
>>> pipe = pipeline.Pipeline([('impute',  
impute), ('scaler', scaler)])
```

Take a look at the Pipeline. As we can see, Pipeline defines the steps that designate the progression of methods:

```
>>> pipe  
Pipeline(steps=[('impute', Imputer(axis=0,  
copy=True, missing_values='NaN',  
strategy='mean', verbose=0)), ('scaler',  
StandardScaler(copy=True, with_mean=True,  
with_std=True))])
```

This is the best part; simply call the `fit_transform` method on the `pipe` object. These separate steps are completed in a single step:

```
>>> new_mat = pipe.fit_transform(mat)  
>>> new_mat [:4, :4]  
array([[ 2.235e+00, -6.291e-01,  1.427e-16,  
-7.496e-01],  
       [ 0.000e+00,  1.158e+00, -9.309e-01,  
9.072e-01],  
       [ 1.068e+00, -2.301e+00,  2.545e+00,
```

```
-2.323e+00],  
[ -1.142e+00,    5.721e-01,   -5.405e-01,  
9.650e-01]])
```

We can also confirm that the two different methods give the same result:

```
>>> np.array_equal(new_mat, mat_imp_and_scaled)  
True
```

Beautiful!

Later in the book, we'll see just how powerful this concept is. It doesn't stop at preprocessing steps. It can easily extend to dimensionality reduction as well, fitting different learning methods. Dimensionality reduction is handled on its own in the recipe *Reducing dimensionality with PCA*.

How it works...

As mentioned earlier, almost every scikit-learn has a similar interface. The important ones that allow Pipelines to function are:

- `fit`
- `transform`
- `fit_transform` (a convenience method)

To be specific, if a Pipeline has N objects, the first $N-1$ objects must implement both `fit` and `transform`, and the N th object must implement at least `fit`. If this doesn't happen, an error will be thrown.

Pipeline will work correctly if these conditions are met, but it is still possible that not every method will work properly. For example, `pipe` has a method, `inverse_transform`, which does exactly what the name entails. However, because the `imputer` step doesn't have an `inverse_transform` method, this method call will fail:

```
>>> pipe.inverse_transform(new_mat)
AttributeError: 'Imputer' object has no
attribute 'inverse_transform'
```

However, this is possible with the `scalar` object:

```
>>> scaler.inverse_transform(new_mat) [:4, :4]
array([[ 0.567, -0.203, -0.806, -0.226],
       [ 0.042,  1.989, -2.254,  1.27 ],
       [ 0.293, -2.254,  3.155, -1.647],
      [-0.226,  1.27 , -1.647,  1.322]])
```

Once a proper Pipeline is set up, it functions almost exactly how you'd expect. It's a series of `for` loops that fit and transform at each intermediate step, feeding the output to the subsequent transformation.

To conclude this recipe, I'll try to answer the "why?" question. There are two main reasons:

- The first reason is **convenience**. The code becomes quite a bit cleaner; instead of calling `fit` and `transform` over and over, it is offloaded to `sklearn`.
- The second, and probably the more important, reason is **cross validation**. Models can become very complex. If a single step in Pipeline has tuning parameters, they might need to be tested; with a single step, the code overhead to test the parameters is low. However, five steps with all of their respective parameters can become difficult to test. Pipelines ease a lot of the burden.

Reducing dimensionality with PCA

Now it's time to take the math up a level! **Principal component analysis (PCA)** is the first somewhat advanced technique discussed in this book. While everything else thus far has been simple statistics, PCA will combine statistics and linear algebra to produce a preprocessing step that can help to reduce dimensionality, which can be the enemy of a simple model.

Getting ready

PCA is a member of the decomposition module of scikit-learn. There are several other decomposition methods available, which will be covered later in this recipe.

Let's use the `iris` dataset, but it's better if you use your own data:

```
>>> from sklearn import datasets  
>>> iris = datasets.load_iris()  
>>> iris_X = iris.data
```

How to do it...

First, import the decomposition module:

```
>>> from sklearn import decomposition
```

Next, instantiate a default PCA object:

```
>>> pca = decomposition.PCA()  
>>> pca  
PCA(copy=True, n_components=None, whiten=False)
```

Compared to other objects in scikit-learn, PCA takes relatively few arguments. Now that the PCA object is created, simply transform the data by calling the `fit_transform` method, with `iris_X` as the argument:

```
>>> iris_pca = pca.fit_transform(iris_X)  
>>> iris_pca[:5]  
array([[ -2.684e+00,   -3.266e-01,    2.151e-02,  
1.006e-03],  
       [ -2.715e+00,    1.696e-01,    2.035e-01,  
9.960e-02],  
       [ -2.890e+00,    1.373e-01,   -2.471e-02,  
1.930e-02],  
       [ -2.746e+00,    3.111e-01,   -3.767e-02,  
-7.596e-02],  
       [ -2.729e+00,   -3.339e-01,   -9.623e-02,  
-6.313e-02]])
```

Now that the PCA has been fit, we can see how well it has

done at explaining the variance (explained in the following *How it works...* section):

```
>>> pca.explained_variance_ratio_
array([ 0.925,  0.053,  0.017,  0.005])
```

How it works...

PCA has a general mathematic definition and a specific use case in data analysis. PCA finds the set of orthogonal directions that represent the original data matrix.

Generally, PCA works by mapping the original dataset into a new space where the new column vectors of the matrix are each orthogonal. From a data analysis perspective, PCA transforms the covariance matrix of the data into column vectors that can "explain" certain percentages of the variance. For example, with the `iris` dataset, 92.5 percent of the variance of the overall dataset can be explained by the first component.

This is extremely useful because dimensionality is problematic in data analysis. Quite often, algorithms applied to high-dimensional datasets will overfit on the initial training, and thus loose generality to the test set. If most of the underlying structure of the data can be faithfully represented by fewer dimensions, then it's generally considered a worthwhile trade-off.

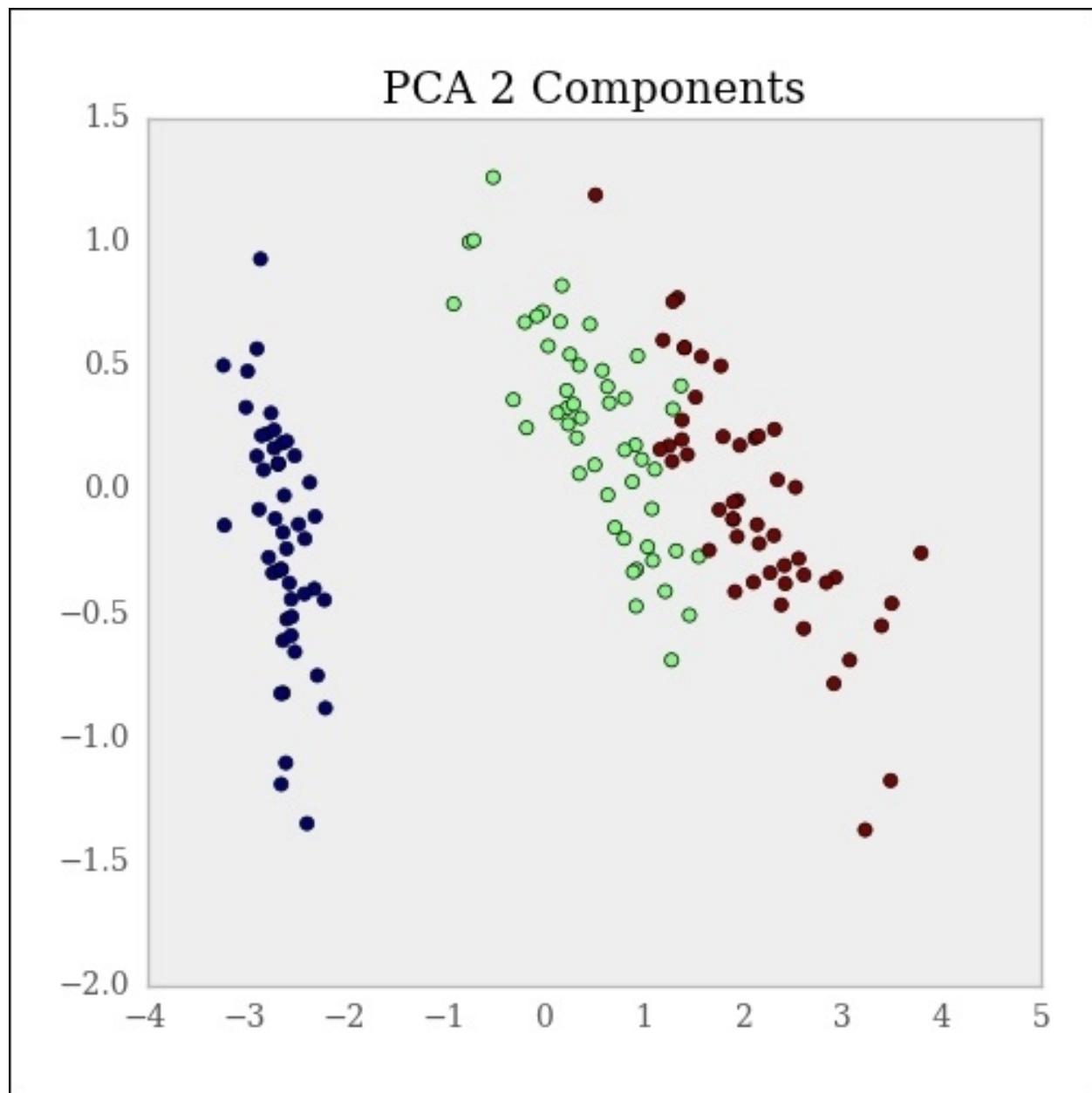
To demonstrate this, we'll apply the PCA transformation to the `iris` dataset and only include two dimensions. The `iris` dataset can normally be separated quite well using

all the dimensions:

```
>>> pca = decomposition.PCA(n_components=2)
>>> iris_X_prime = pca.fit_transform(iris_X)
>>> iris_X_prime.shape
(150, 2)
```

Our data matrix is now 150 x 2, instead of 150 x 4.

The usefulness of two dimensions is that it is now very easy to plot.



The separability of the classes remain even after reducing the number of dimensionality by two.

We can see how much of the variance is represented by

the two components that remain:

```
>>> pca.explained_variance_ratio_.sum()  
0.9776
```

There's more...

The PCA object can also be created with the amount of explained variance in mind from the start. For example, if we want to be able to explain at least 98 percent of the variance, the PCA object will be created as follows:

```
>>> pca = decomposition.PCA(n_components=.98)
>>> iris_X_prime = pca.fit(iris_X)
>>> pca.explained_variance_ratio_.sum()
1.0
```

Since we wanted to explain variance slightly more than the two component examples, a third was included.

Using factor analysis for decomposition

Factor analysis is another technique we can use to reduce dimensionality. However, factor analysis makes assumptions and PCA does not. The basic assumption is that there are implicit features responsible for the features of the dataset.

This recipe will boil down to the explicit features from our samples in an attempt to understand the independent variables as much as the dependent variables.

Getting ready

To compare PCA and factor analysis, let's use the `iris` dataset again, but we'll first need to load the factor analysis class:

```
>>> from sklearn.decomposition import  
FactorAnalysis
```

How to do it...

From a programming perspective, factor analysis isn't much different from PCA:

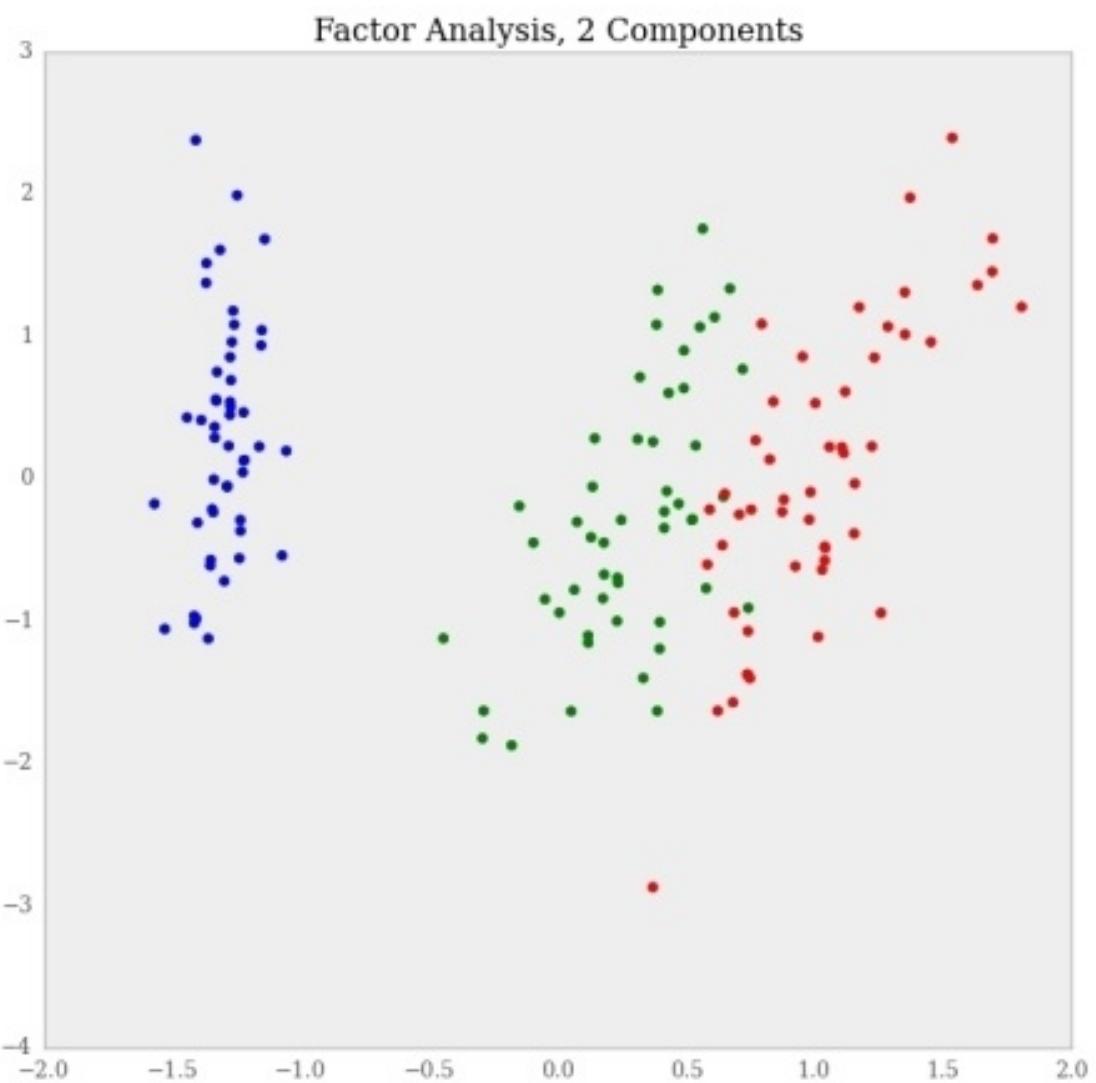
```
>>> fa = FactorAnalysis(n_components=2)
>>> iris_two_dim = fa.fit_transform(iris.data)
>>> iris_two_dim[:5]
array([[-1.33125848,  0.55846779],
       [-1.33914102, -0.00509715],
       [-1.40258715, -0.307983],
       [-1.29839497, -0.71854288],
       [-1.33587575,  0.36533259]])
```

Tip

Downloading the example code

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

Compare the following plot to the plot in the last section:



Since factor analysis is a probabilistic transform, we can examine different aspects such as the log likelihood of the observations under the model, and better still, compare the log likelihoods across models.

Factor analysis is not without flaws. The reason is that you're not fitting a model to predict an outcome, you're fitting a model as a preparation step. This isn't a bad thing per se, but errors here compound when training the actual model.

How it works...

Factor analysis is similar to PCA, which was covered previously. However, there is an important distinction to be made. PCA is a linear transformation of the data to a different space where the first component "explains" the variance of the data, and each subsequent component is orthogonal to the first component.

For example, you can think of PCA as taking a dataset of N dimensions and going down to some space of M dimensions, where $M < N$.

Factor analysis, on the other hand, works under the assumption that there are only M important features and a linear combination of these features (plus noise) creates the dataset in N dimensions. To put it another way, you don't do regression on an outcome variable, you do regression on the features to determine the latent factors of the dataset.

Kernel PCA for nonlinear dimensionality reduction

Most of the techniques in statistics are linear by nature, so in order to capture nonlinearity, we might need to apply some transformation. PCA is, of course, a linear transformation. In this recipe, we'll look at applying nonlinear transformations, and then apply PCA for dimensionality reduction.

Getting ready

Life would be so easy if data was always linearly separable, but unfortunately it's not. Kernel PCA can help to circumvent this issue. Data is first run through the kernel function that projects the data onto a different space; then PCA is performed.

To familiarize yourself with the kernel functions, it will be a good exercise to think of how to generate data that is separable by the kernel functions available in the kernel PCA. Here, we'll do that with the cosine kernel. This recipe will have a bit more theory than the previous recipes.

How to do it...

The **cosine kernel** works by comparing the angle between two samples represented in the feature space. It is useful when the magnitude of the vector perturbs the typical distance measure used to compare samples.

As a reminder, the cosine between two vectors is given by the following:

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

This means that the cosine between A and B is the dot product of the two vectors normalized by the product of the individual norms. The magnitude of vectors A and B have no influence on this calculation.

So, let's generate some data and see how useful it is. First, we'll imagine there are two different underlying processes; we'll call them A and B:

```
>>> import numpy as np  
>>> A1_mean = [1, 1]  
>>> A1_cov = [[2, .99], [1, 1]]  
>>> A1 = np.random.multivariate_normal(A1_mean,
```

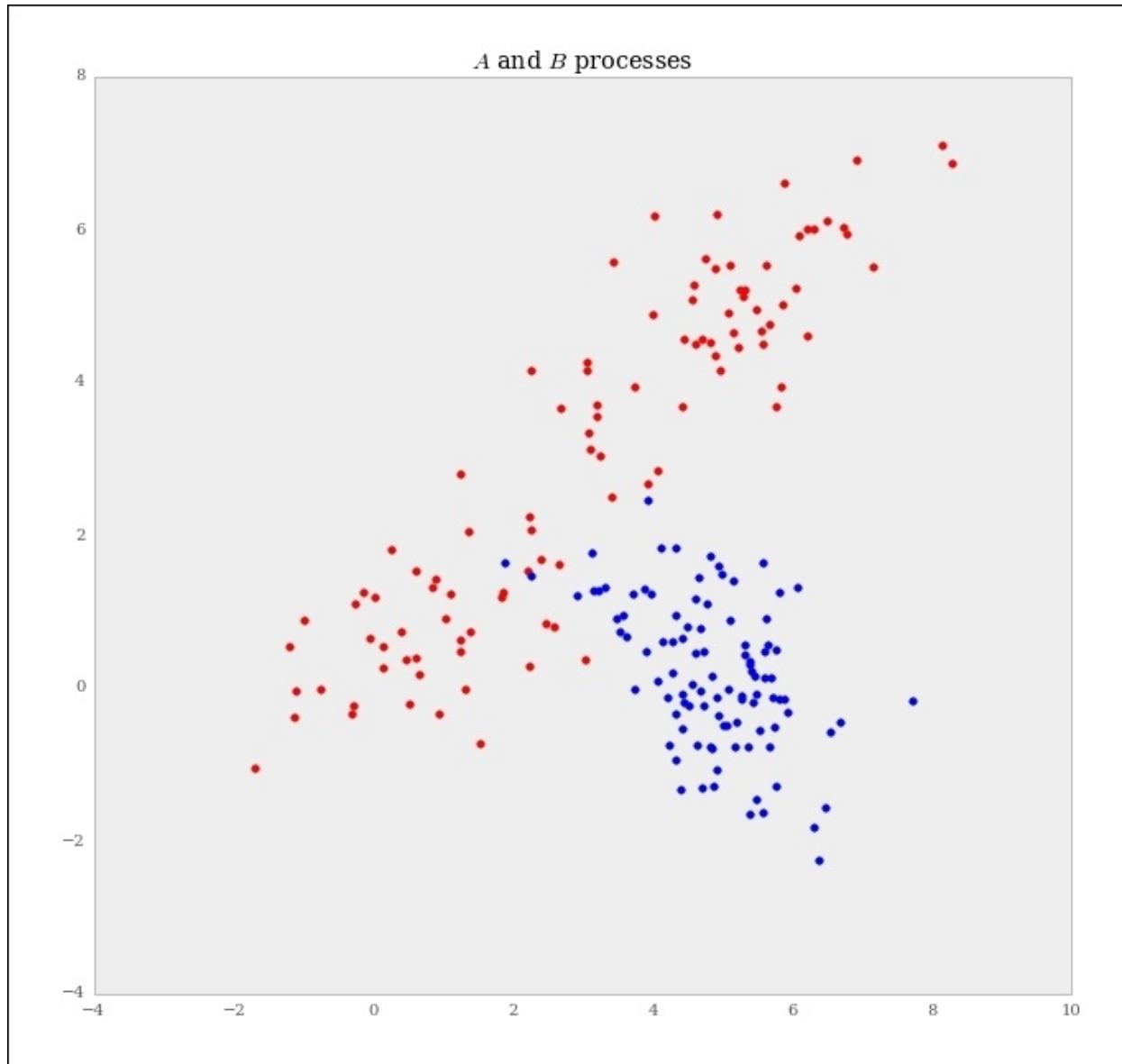
```
A1_cov, 50)

>>> A2_mean = [5, 5]
>>> A2_cov = [[2, .99], [1, 1]]
>>> A2 = np.random.multivariate_normal(A2_mean,
A2_cov, 50)

>>> A = np.vstack((A1, A2))

>>> B_mean = [5, 0]
>>> B_cov = [[.5, -1], [-.9, .5]]
>>> B = np.random.multivariate_normal(B_mean,
B_cov, 100)
```

Once plotted, it will look like the following:

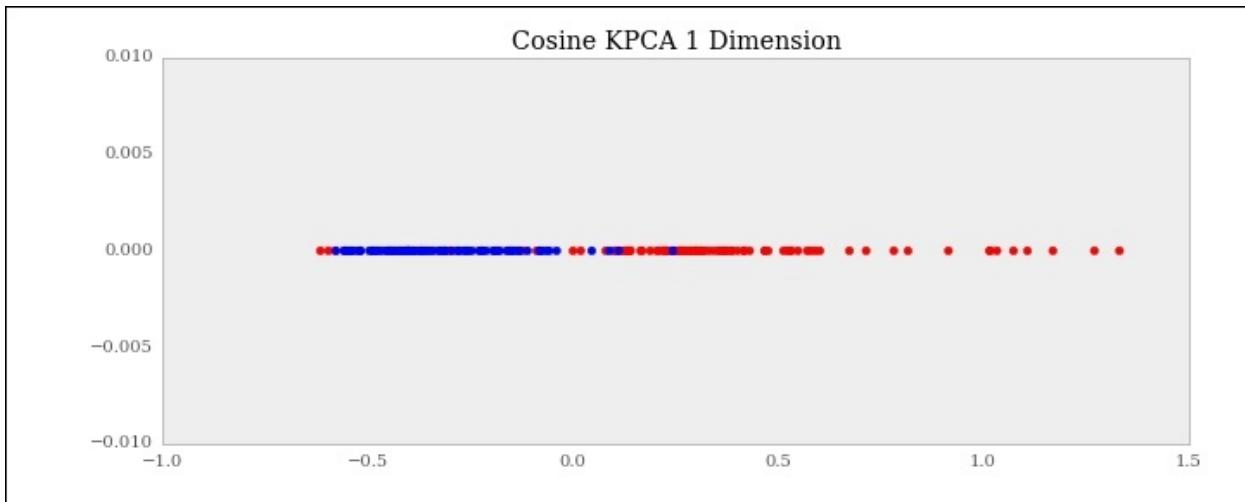


By visual inspection, it seems that the two classes are from different processes, but separating them in one slice might be difficult. So, we'll use the kernel PCA with the cosine kernel discussed earlier:

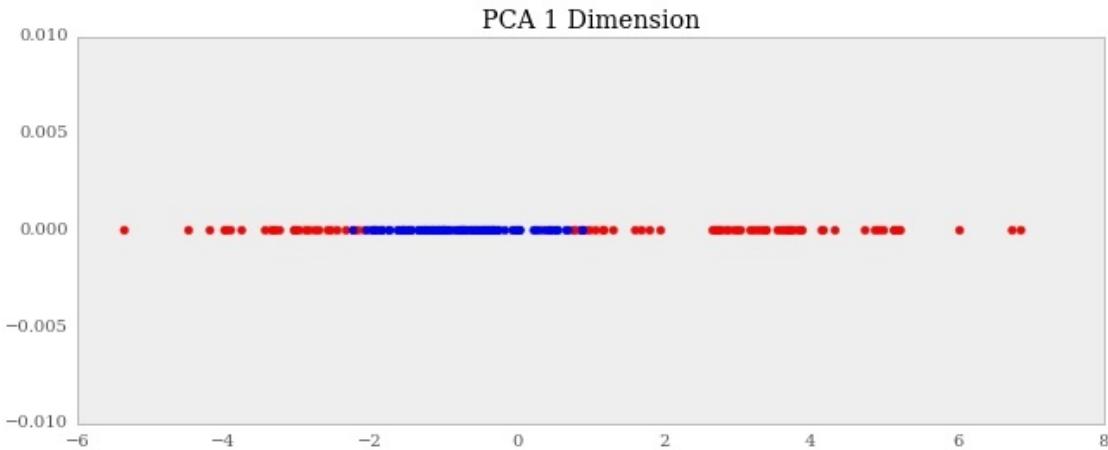
```
>>> kpca =
```

```
decomposition.KernelPCA(kernel='cosine',  
n_components=1)  
>>> AB = np.vstack((A, B))  
>>> AB_transformed = kpca.fit_transform(AB)
```

Visualized in one dimension after the kernel PCA, the dataset looks like the following:



Contrast this with PCA without a kernel:



Clearly, the kernel PCA does a much better job.

How it works...

There are several different kernels available as well as the cosine kernel. You can even write your own kernel function. The available kernels are:

- `poly` (polynomial)
- `rbf` (radial basis function)
- `sigmoid`
- `cosine`
- `precomputed`

There are also options contingent of the kernel choice. For example, the `degree` argument will specify the degree for the `poly`, `rbf`, and `sigmoid` kernels; also, `gamma` will affect the `rbf` or `poly` kernels.

The recipe on SVM will cover the `rbf` kernel function in more detail.

A word of caution: kernel methods are great to create separability, but they can also cause overfitting if used without care.

Using truncated SVD to reduce dimensionality

Truncated **Singular Value Decomposition (SVD)** is a matrix factorization technique that factors a matrix M into the three matrices U , Σ , and V . This is very similar to PCA, excepting that the factorization for SVD is done on the data matrix, whereas for PCA, the factorization is done on the covariance matrix. Typically, SVD is used under the hood to find the principle components of a matrix.

Getting ready

Truncated SVD is different from regular SVDs in that it produces a factorization where the number of columns is equal to the specified truncation. For example, given an $n \times n$ matrix, SVD will produce matrices with n columns, whereas truncated SVD will produce matrices with the specified number of columns. This is how the dimensionality is reduced.

Here, we'll again use the `iris` dataset so that you can compare this outcome against the PCA outcome:

```
>>> from sklearn.datasets import load_iris  
>>> iris = load_iris()  
>>> iris_data = iris.data  
>>> iris_target = iris.target
```

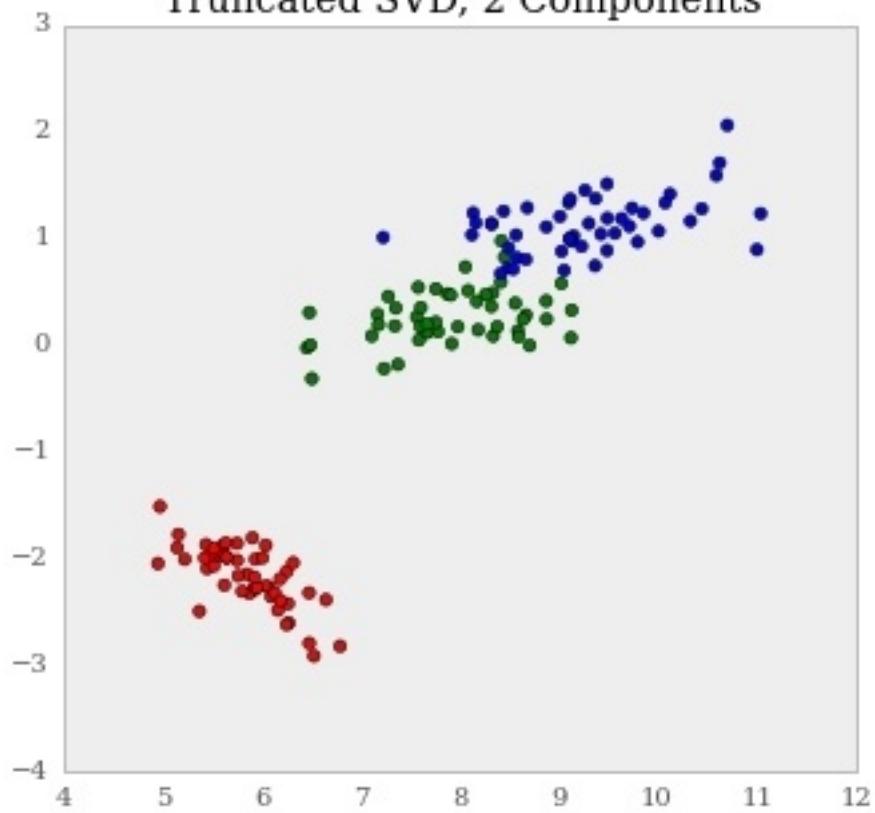
How to do it...

This object follows the same form as the other objects we've used. First, we'll import the required object, then we'll fit the model and examine the results:

```
>>> from sklearn.decomposition import  
TruncatedSVD  
>>> svd = TruncatedSVD(2)  
>>> iris_transformed =  
svd.fit_transform(iris_data)  
>>> iris_data[:5]  
array([[ 5.1,  3.5,  1.4,  0.2],  
       [ 4.9,  3. ,  1.4,  0.2],  
       [ 4.7,  3.2,  1.3,  0.2],  
       [ 4.6,  3.1,  1.5,  0.2],  
       [ 5. ,  3.6,  1.4,  0.2]])  
  
>>> iris_transformed[:5]  
array([[ 5.91220352, -2.30344211],  
       [ 5.57207573, -1.97383104],  
       [ 5.4464847 , -2.09653267],  
       [ 5.43601924, -1.87168085],  
       [ 5.87506555, -2.32934799]])
```

The output will look like the following:

Truncated SVD, 2 Components



How it works...

Now that we've walked through how TruncatedSVD is performed in scikit-learn, let's look at how we can use only `scipy`, and learn a bit in the process.

First, we need to use `linalg` of `scipy` to perform SVD:

```
>>> from scipy.linalg import svd
>>> D = np.array([[1, 2], [1, 3], [1, 4]])
>>> D
array([[1, 2],
       [1, 3],
       [1, 4]])

>>> U, S, V = svd(D, full_matrices=False)
>>> U.shape, S.shape, V.shape
((3, 2), (2,), (2, 2))
```

We can reconstruct the original matrix D to confirm U, S, and V as a decomposition:

```
>>> np.dot(U.dot(np.diag(S)), V)
array([[1, 2],
       [1, 3],
       [1, 4]])
```

The matrix that is actually returned by TruncatedSVD is the dot product of the U and S matrices.

If we want to simulate the truncation, we will drop the smallest singular values and the corresponding column vectors of U . So, if we want a single component here, we do the following:

```
>>> new_S = S[0]
>>> new_U = U[:, 0]
>>> new_U.dot(new_S)
array([-2.20719466, -3.16170819, -4.11622173])
```

In general, if we want to truncate to some dimensionality, for example, t , we drop $N-t$ singular values.

There's more...

`TruncatedSVD` has a few miscellaneous things that are worth noting with respect to the method.

Sign flipping

There's a "gotcha" with truncated SVDs. Depending on the state of the random number generator, successive fittings of `TruncatedSVD` can flip the signs of the output. In order to avoid this, it's advisable to fit `TruncatedSVD` once, and then use transforms from then on. Another good reason for Pipelines!

To carry this out, do the following:

```
>>> tsvd = TruncatedSVD(2)
>>> tsvd.fit(iris_data)
>>> tsvd.transform(iris_data)
```

Sparse matrices

One advantage of `TruncatedSVD` over PCA is that `TruncatedSVD` can operate on sparse matrices while PCA cannot. This is due to the fact that the covariance matrix must be computed for PCA, which requires operating on the entire matrix.

Decomposition to classify with DictionaryLearning

In this recipe, we'll show how a decomposition method can actually be used for classification.

`DictionaryLearning` attempts to take a dataset and transform it into a sparse representation.

Getting ready

With `DictionaryLearning`, the idea is that the features are a basis for the resulting datasets. In an effort to keep this recipe short, I'll assume you have `iris_data` and `iris_target` ready to go.

How to do it...

First, import DictionaryLearning:

```
>>> from sklearn.decomposition import  
DictionaryLearning
```

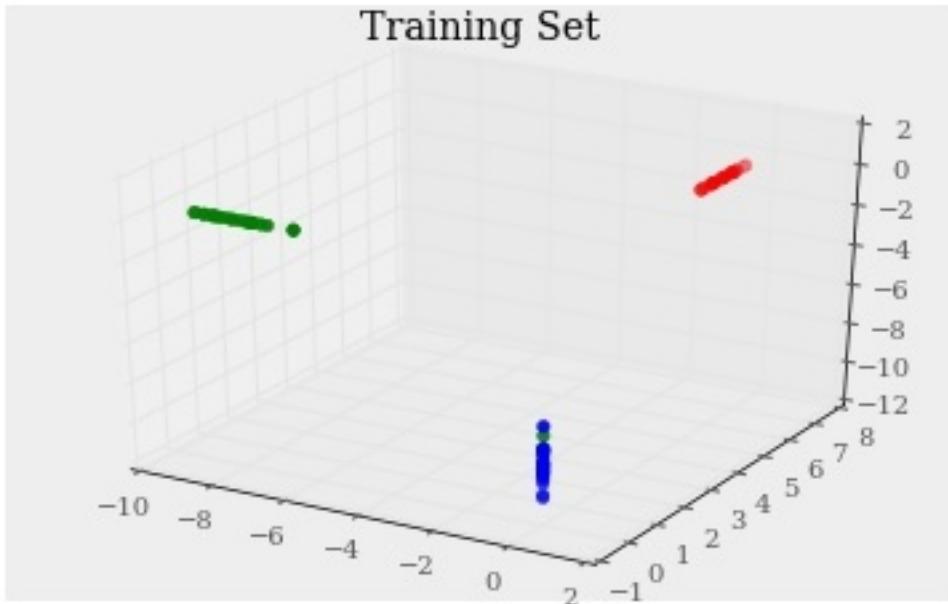
Next, use three components to represent the three species of iris:

```
>>> dl = DictionaryLearning(3)
```

Then transform every other data point so that we can test the classifier on the resulting data points after the learner is trained:

```
>>> transformed =  
dl.fit_transform(iris_data[::2])  
>>> transformed[:5]  
array([[ 0.           ,  6.34476574,   0.           ],  
       [ 0.           ,  5.83576461,   0.           ],  
       [ 0.           ,  6.32038375,   0.           ],  
       [ 0.           ,  5.89318572,   0.           ],  
       [ 0.           ,  5.45222715,   0.           ]])
```

We can visualize the output. Notice how each value is sited on the x , y , or z axis along with the other values and 0; this is called sparseness.

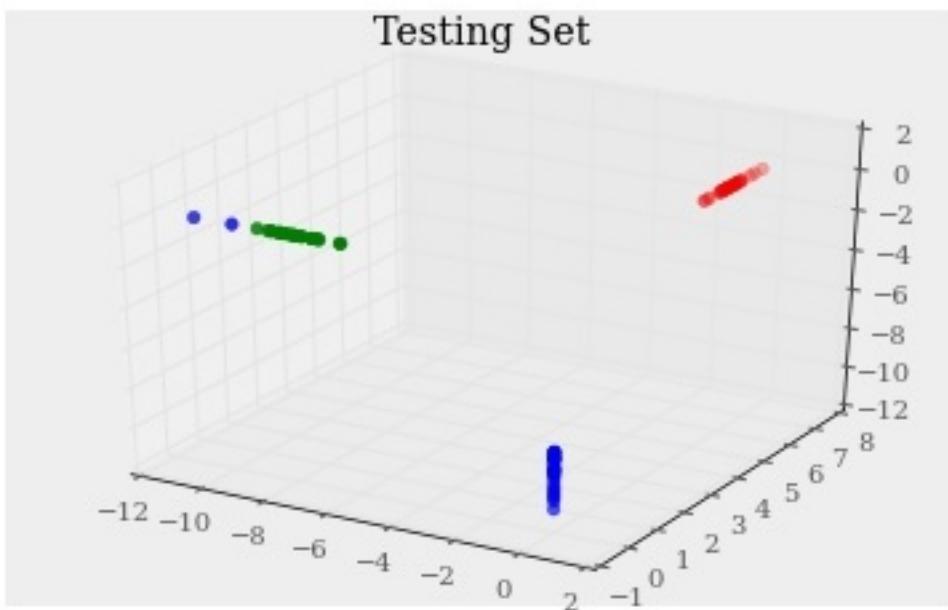


If you look closely, you can see there was some training error. One of the classes was misclassified. Only being wrong once isn't a big deal, though.

Next, let's fit (not `fit_transform`) the testing set:

```
>>> transformed = dl.transform(iris_data[1::2])
```

The following screenshot shows its performance:



Notice again that there was some error in the classification. If you remember some of the other visualizations, the blue and green classes were the two classes that often appeared close together.

How it works...

DictionaryLearning has a background in signal processing and neurology. The idea is that only few features can be active at any given time. Therefore, DictionaryLearning attempts to find a suitable representation for the underlying data, given the constraint that most of the features should be 0.

Putting it all together with Pipelines

Now that we've used Pipelines and data transformation techniques, we'll walk through a more complicated example that combines several of the previous recipes into a pipeline.

Getting ready

In this section, we'll show off some more of Pipeline's power. When we used it earlier to impute missing values, it was only a quick taste; we'll chain together multiple preprocessing steps to show how Pipelines can remove extra work.

Let's briefly load the `iris` dataset and seed it with some missing values:

```
>>> from sklearn.datasets import load_iris
>>> import numpy as np

>>> iris = load_iris()
>>> iris_data = iris.data

>>> mask = np.random.binomial(1, .25,
iris_data.shape).astype(bool)
>>> iris_data[mask] = np.nan
>>> iris_data[:5]
array([[ 5.1,  3.5,  1.4,  nan],
       [ nan,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  nan,  0.2]])
```

How to do it...

The goal of this chapter is to first impute the missing values of `iris_data`, and then perform PCA on the corrected dataset. You can imagine (and we'll do it later) that this workflow might need to be split between a training dataset and a holdout set; Pipelines will make this easier, but first we need to take a baby step.

Let's load the required libraries:

```
>>> from sklearn import pipeline, preprocessing,  
decomposition
```

Next, create the imputer and PCA classes:

```
>>> pca = decomposition.PCA()  
>>> imputer = preprocessing.Imputer()
```

Now that we have the classes we need, we can load them into Pipeline:

```
>>> pipe = pipeline.Pipeline([('imputer',  
imputer), ('pca', pca)])  
>>> iris_data_transformed =  
pipe.fit_transform(iris_data)  
>>> iris_data_transformed[:5]  
array([[ -2.42e+00,  -3.59e-01,  -6.88e-01,  
-3.49e-01],  
[ -2.44e+00,  -6.94e-01,   3.27e-01,
```

```
4.87e-01],  
    [-2.94e+00, 2.45e-01, -1.85e-03,  
4.37e-02],  
    [-2.79e+00, 4.29e-01, -8.05e-03,  
9.65e-02],  
    [-6.46e-01, 8.87e-01, 7.54e-01,  
-5.19e-01]])
```

This takes a lot more management if we use separate steps. Instead of each step requiring a fit transform, this step is performed only once. Not to mention that we only have to keep track of one object!

How it works...

Hopefully it was obvious, but each step in Pipeline is passed to a Pipeline object via a list of tuples, with the first element getting the name and the second getting the actual object.

Under the hood, these steps are looped through when a method such as `fit_transform` is called on the Pipeline object.

This said, there are quick and dirty ways to create Pipeline, much in the same way there was a quick way to perform scaling, though we can use `StandardScaler` if we want more power. The `pipeline` function will automatically create the names for the Pipeline objects:

```
>>> pipe2 = pipeline.make_pipeline(imputer, pca)
>>> pipe2.steps
[('imputer', Imputer(axis=0, copy=True,
missing_values='NaN', strategy='mean',
verbose=0)),
 ('pca', PCA(copy=True, n_components=None,
whiten=False))]
```

This is the same object that was created in the more verbose method:

```
>>> iris_data_transformed2 =
```

```
pipe2.fit_transform(iris_data)
>>> iris_data_transformed2[:5]
array([[ -2.42e+00,  -3.59e-01,  -6.88e-01,
       -3.49e-01],
       [ -2.44e+00,  -6.94e-01,   3.27e-01,
       4.87e-01],
       [ -2.94e+00,   2.45e-01,  -1.85e-03,
       4.37e-02],
       [ -2.79e+00,   4.29e-01,  -8.05e-03,
       9.65e-02],
       [ -6.46e-01,   8.87e-01,   7.54e-01,
      -5.19e-01]])
```

There's more...

We just walked through Pipelines at a very high level, but it's unlikely that we will want to apply the base transformation. Therefore, the attributes of each object in Pipeline can be accessed from a `set_params` method, where the parameter follows the `<parameter>'s_name>__<parameter>'s_parameter>` convention. For example, let's change the `pca` object to use two components:

```
>>> pipe2.set_params(pca_n_components=2)
Pipeline(steps=[('imputer', Imputer(axis=0,
copy=True,
missing_values='NaN', strategy='mean',
verbose=0)),
               ('pca', PCA(copy=True, n_components=2,
whiten=False))])
```

Tip

The `_` notation is pronounced as **dunder** in the Python community.

Notice `n_components=2` in the preceding output. Just as a test, we can output the same transformation we already did twice, and the output will be an $N \times 2$ matrix:

```
>>> iris_data_transformed3 =
```

```
pipe2.fit_transform(iris_data)
>>> iris_data_transformed3[:5]
array([[-2.42, -0.36],
       [-2.44, -0.69],
       [-2.94,  0.24],
       [-2.79,  0.43],
       [-0.65,  0.89]])
```

Using Gaussian processes for regression

In this recipe, we'll use the **Gaussian process** for regression. In the linear models section, we saw how representing prior information on the coefficients was possible using **Bayesian Ridge Regression**.

With a Gaussian process, it's about the variance and not the mean. However, with a Gaussian process, we assume the mean is 0, so it's the covariance function we'll need to specify.

The basic setup is similar to how a prior can be put on the coefficients in a typical regression problem. With a GP, a prior can be put on the functional form of the data, and it's the covariance between the data points that is used to model the data, and therefore, must be fit from the data.

Getting ready

So, let's use some regression data and walkthrough how Gaussian processes work in scikit-learn:

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()

>>> boston_X = boston.data
>>> boston_y = boston.target

>>> train_set = np.random.choice([True, False] ,
len(boston_y) ,
p=[.75, .25])
```

How to do it...

Now that we have the data, we'll create a scikit-learn GaussianProcess object. By default, it uses a constant regression function and squared exponential correlation, which is one of the more common choices:

```
>>> from sklearn.gaussian_process import  
GaussianProcess  
>>> gp = GaussianProcess()  
>>> gp.fit(boston_X[train_set],  
boston_y[train_set])  
GaussianProcess(beta0=None, corr=<function  
squared_exponential  
    at 0x110809488>, normalize=True,  
    nugget=array(2.220446049250313e-  
15),  
    optimizer='fmin_cobyla',  
    random_start=1,  
    random_state=<mtrand.RandomState  
object  
    at 0x10b9b58b8>, regr=<function  
constant  
    at 0x1108090c8>,  
    storage_mode='full',  
    theta0=array([[ 0.1]]),  
    thetaL=None, thetaU=None,  
    verbose=False)
```

That's a formidable object definition. The following are a couple of things to point out:

- `beta0`: This is the regression weight. This defaults in a way such that MLE is used for estimation.
- `corr`: This is the correlation function. There are several built-in correlation functions. We'll look at more of them in the following *How it works...* section.
- `regr`: This is the constant regression function.
- `nugget`: This is the regularization parameter. It defaults to a very small number. You can either pass one value to be used for each data point or a single value that needs to be applied uniformly.
- `normalize`: This defaults to `True`, and it will center and scale the features. This would be `scale` in R.

Okay, so now that we fit the object, let's look at its performance against the test object:

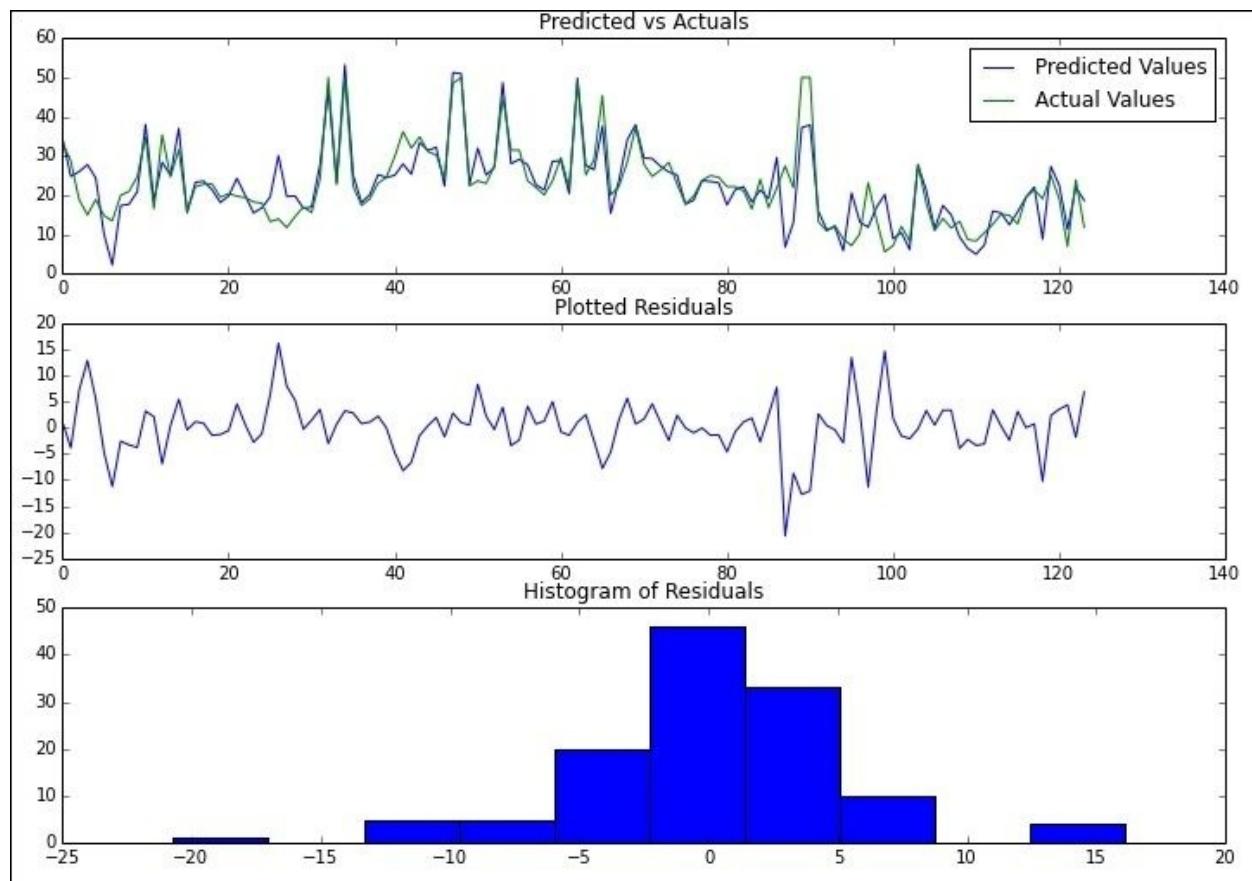
```
>>> test_preds =
gp.predict(boston_X[~train_set])
```

Let's plot the predicted values versus the actual values; then, because we're doing regression, it's probably a good idea to look at plotted residuals and a histogram of the residuals:

```
>>> from matplotlib import pyplot as plt
>>> f, ax = plt.subplots(figsize=(10, 7),
nrows=3)
>>> f.tight_layout()
```

```
>>> ax[0].plot(range(len(test_preds)),  
test_preds,  
                label='Predicted Values');  
>>> ax[0].plot(range(len(test_preds)),  
boston_y[~train_set],  
                label='Actual Values');  
>>> ax[0].set_title("Predicted vs Actuals")  
>>> ax[0].legend(loc='best')  
  
>>> ax[1].plot(range(len(test_preds)),  
                test_preds -  
boston_y[~train_set]);  
>>> ax[1].set_title("Plotted Residuals")  
  
>>> ax[2].hist(test_preds -  
boston_y[~train_set]);  
>>> ax[2].set_title("Histogram of Residuals")
```

The output is as follows:



How it works...

Now that we've worked through a very quick example, let's look a little more at what some of the parameters do and how we can tune them based on the model we're trying to fit.

First, let's try to understand what's going on with the `corr` function. This function describes the relationship between the different pairs of X . The following five different correlation functions are offered by scikit-learn:

- `absolute_exponential`
- `squared_exponential`
- `generalized_exponential`
- `cubic`
- `linear`

For example, the squared exponential has the following form:

$$K = \exp\left(-\left\{\frac{|d|^2}{2l^2}\right\}\right)$$

Linear, on the other hand, is just the dot product of the

two points in question:

$$K = x^T x \{ \cdot \}$$

Another parameter of interest is `theta0`. This represents the starting point in the estimation of the parameters.

Once we have an estimation of K and the mean, the process is fully specified due to it being a Gaussian process; emphasis is put on Gaussian, a reason it's so popular for general machine learning work.

Let's use a different `regr` function, apply a different `theta0`, and look at how the predictions differ:

```
>>> gp = GaussianProcess(regr='linear',
   theta0=5e-1)
>>> gp.fit(boston_X[train_set],
   boston_y[train_set]);
>>> linear_preds =
gp.predict(boston_X[~train_set])
>>> f, ax = plt.subplots(figsize=(7, 5))
```

Let's have a look at the fit:

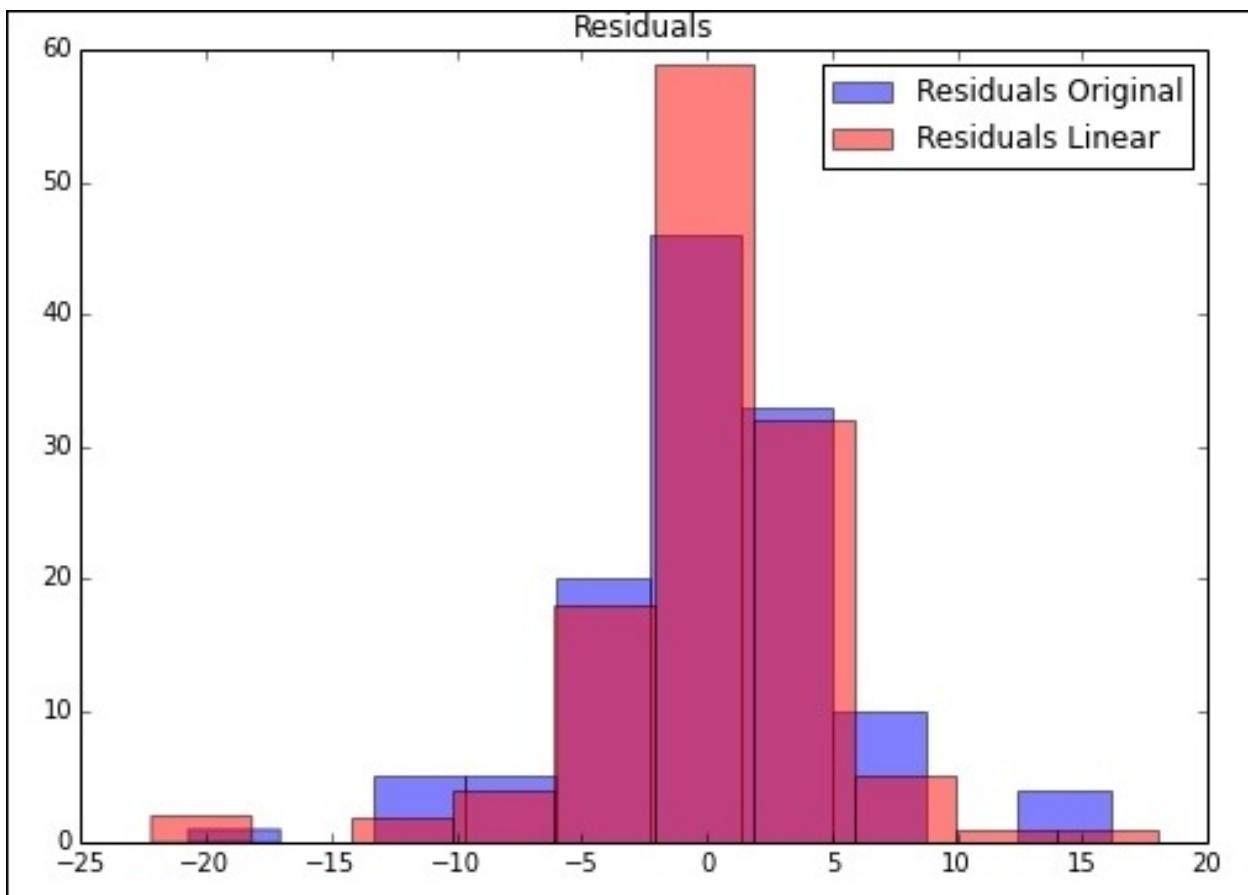
```
>>> f.tight_layout()
>>> ax.hist(test_preds - boston_y[~train_set],
```

```

        label='Residuals Original' ,
color='b' , alpha=.5);
>>> ax.hist(linear_preds - boston_y[~train_set] ,
           label='Residuals Linear' , color='r' ,
alpha=.5);
>>> ax.set_title("Residuals")
>>> ax.legend(loc='best')

```

The following is the output:



Clearly, the second model's predictions are slightly better

for the most part. If we want to sum this up, we can look at the MSE of the predictions:

```
>>> np.power(test_preds - boston_y[~train_set],  
2).mean()  
26.254844099612455  
>>> np.power(linear_preds -  
boston_y[~train_set], 2).mean()  
21.938924337056068
```

There's more...

We might want to understand the uncertainty in our estimates. When we make the predictions, if we pass the `eval_MSE` argument as `True`, we'll get MSE and the predicted values. From a mechanics standpoint, a tuple of predictions and MSE is returned:

```
>>> test_preds, MSE =
gp.predict(boston_X[~train_set], eval_MSE=True)
>>> MSE[:5]
array([ 11.95314572,    8.48397825,    6.0287539 ,
29.20844347,
       0.36427829])
```

So, now that we have errors in the estimates (unfortunately), let's plot the first few to get an indication of accuracy:

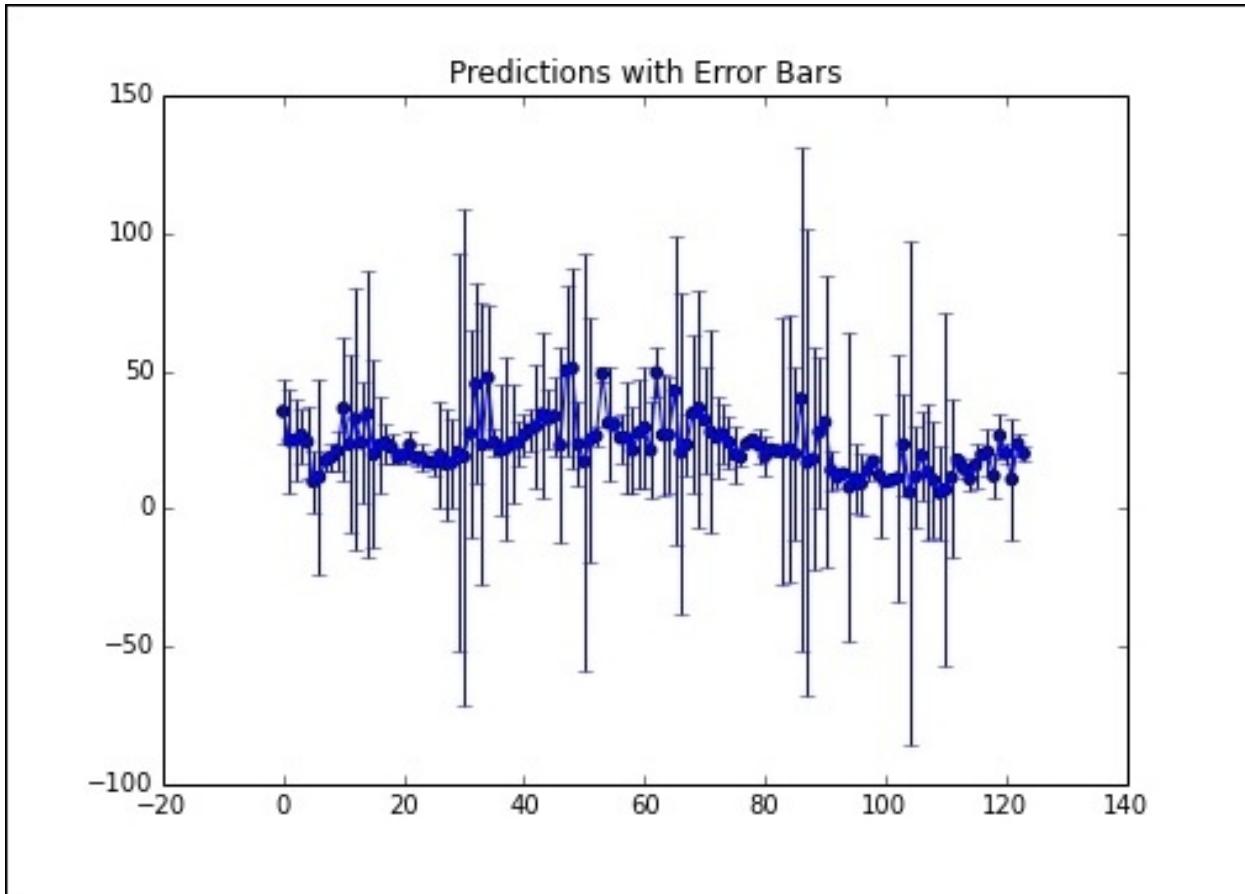
```
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> n = 20
>>> rng = range(n)
>>> ax.scatter(rng, test_preds[:n])
>>> ax.errorbar(rng, test_preds[:n],
yerr=1.96*MSE[:n])

>>> ax.set_title("Predictions with Error Bars")

>>> ax.set_xlim((-1, 21));
```

The following is the output:



As you can see, there's quite a bit of variance in the estimates for a lot of these points. On the other hand, our overall error wasn't too bad.

Defining the Gaussian process object directly

We just touched the surface of Gaussian processes. In this recipe, we'll look at how we can directly access the Gaussian process object with the correlation function we want.

Getting ready

Within the `gaussian_process` module, there is direct access to many of the correlation functions or regression functions. This means that instead of creating the `GaussianProcess` object, we can just create this object through a function. If you're more familiar with object-oriented code, this is basically a class method at the module level.

In this chapter, we'll march through most of the functions and show their results on example data. Do not stop at these examples if you want to get more familiar with the behavior of the various covariate functions. Hopefully, you're still using IPython (or the notebook).

Since this doesn't expose anything new mathematically, we'll just show how to do it.

How to do it...

First, we'll import some basic regression data:

```
>>> from sklearn.datasets import make_regression  
>>> X, y = make_regression(1000, 1, 1)  
>>> from sklearn.gaussian_process import  
regression_models
```

First up is the constant correlation function. This will comprise a constant and more for completeness:

```
>>> regression_models.constant(X) [:5]  
array([[ 1.],  
       [ 1.],  
       [ 1.],  
       [ 1.],  
       [ 1.]])
```

Another option is the squared exponential correlation function. This is also the default for the GaussianProcess class:

```
>>> regression_models.linear(X) [:1]  
array([[ 1.,  0.38833572]])  
  
>>> regression_models.quadratic(X) [:1]  
array([[ 1.,  0.38833572,  0.15080463]])
```

How it works...

Now that we have the regression function, we can feed it directly into the `GaussianProcess` object. The default is the constant regression function, but we can just as easily pass it in a linear model or a quadratic model.

Using stochastic gradient descent for regression

In this recipe, we'll get our first taste of stochastic gradient descent. We'll use it for regression here, but for the next recipe, we'll use it for classification.

Getting ready

Stochastic Gradient Descent (SGD) is often an unsung hero in machine learning. Underneath many algorithms, there is SGD doing the work. It's popular due to its simplicity and speed—these are both very good things to have when dealing with a lot of data.

The other nice thing about SGD is that while it's at the core of many ML algorithms computationally, it does so because it easily describes the process. At the end of the day, we apply some transformation on the data, and then we fit our data to the model with some loss function.

How to do it...

If SGD is good on large datasets, we should probably test it on a fairly large dataset:

```
>>> from sklearn import datasets  
>>> X, y = datasets.make_regression(int(1e6))  
# Just in case the 1e6 throws you off.  
>>> print "{:,} ".format(int(1e6))  
1,000,000
```

It's probably worth gaining some intuition about the composition and size of the object. Thankfully, we're dealing with NumPy arrays, so we can just access `nbytes`. The built-in Python way to access the object size doesn't work for NumPy arrays. This output be system dependent, so you may not get the same results:

```
>>> print "{:,} ".format(X nbytes)  
800,000,000
```

To get some human perspective, we can convert `nbytes` to megabytes. There are roughly 1 million bytes in an MB:

```
>>> X nbytes / 1e6  
800.0
```

So, the number of bytes per data point is:

```
>>> x.nbytes / (x.shape[0]*x.shape[1])  
8
```

Well, isn't that tidy, and fairly tangential, for what we're trying to accomplish; however, it's worth knowing how to get the size of the objects you're dealing with.

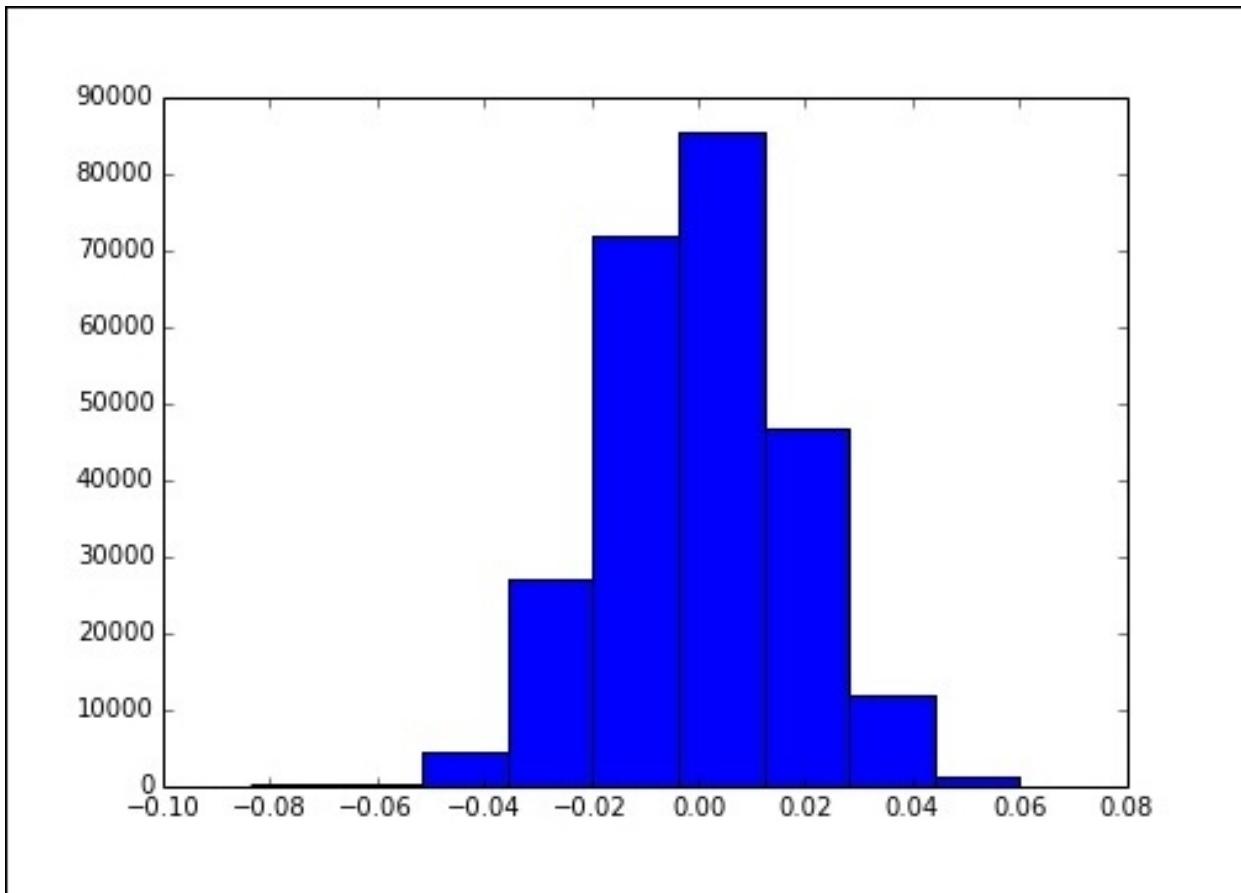
So, now that we have the data, we can simply fit a SGDRegressor model:

```
>>> from sklearn import linear_model  
>>> sgd = linear_model.SGDRegressor()  
>>> train = np.random.choice([True, False],  
size=len(y), p=[.75, .25])  
>>> sgd.fit(X[train], y[train])  
SGDRegressor(alpha=0.0001, epsilon=0.1,  
eta0=0.01,  
            fit_intercept=True, l1_ratio=0.15,  
            learning_rate='invscaling',  
            loss='squared_loss',  
            n_iter=5, penalty='l2',  
            power_t=0.25, random_state=None,  
            shuffle=False, verbose=0,  
            warm_start=False)
```

So, we have another "beefy" object. The main thing to know now is that our loss function is `squared_loss`, which is the same thing that occurs during linear regression. Also worth noting is that `shuffle` will generate a random shuffle of the data. This is useful if you want to break a potentially spurious correlation. With

`fit_intercept`, scikit-learn will automatically include a column of ones. If you like to see more through the output of the fitting, set `verbose` to 1.

We can then predict, as we previously have, using scikit-learn's consistent API:



You can see we actually got a really good fit. There is barely any variation and the histogram has a nice normal look.

How it works...

Clearly, the fake dataset we used wasn't too bad, but you can imagine datasets with larger magnitudes. For example, if you worked in Wall Street on any given day, there might be two billion transactions on any given exchange in a market. Now, imagine that you have a week's or year's data. Running in-core algorithms does not work with huge volumes of data.

The reason this is normally difficult is that to do standard gradient descent, we're required to calculate the gradient at every step. The gradient has the standard definition from any third calculus course.

The gist of the algorithm is that at each step we calculate a new set of coefficients and update this by a learning rate and the outcome of the objective function.

In pseudo code, this might look like the following:

```
>>> while not_converged:  
        w = w - learning_rate*gradient(cost(w))
```

The relevant variables are as follows:

- w : This is the coefficient matrix.
- $learning_rate$: This shows how big a step to take at

each iteration. This might be important to tune if you aren't getting a good convergence.

- `gradient`: This is the matrix of second derivatives.
- `cost`: This is the squared error for regression. We'll see later that this cost function can be adapted to work with classification tasks. This flexibility is one thing that makes SGD so useful.

This will not be so bad, except for the fact that the gradient function is expensive. As the vector of coefficients gets larger, calculating the gradient becomes very expensive. For each update step, we need to calculate a new weight for every point in the data, and then update.

The stochastic gradient descent works slightly differently; instead of the previous definition for batch gradient descent, we'll update the parameter with each new data point. This data point is picked at random, and hence the name stochastic gradient descent.

Chapter 2. Working with Linear Models

In this chapter, we will cover the following topics:

- Fitting a line through data
- Evaluating the linear regression model
- Using ridge regression to overcome linear regression's shortfalls
- Optimizing the ridge regression parameter
- Using sparsity to regularize models
- Taking a more fundamental approach to regularization with LARS
- Using linear methods for classification – logistic regression
- Directly applying Bayesian ridge regression
- Using boosting to learn from errors

Introduction

Linear models are fundamental in statistics and machine learning. Many methods rely on a linear combination of variables to describe the relationship in the data. Quite often, great efforts are taken in an attempt to make the transformations necessary so that the data can be described in a linear combination.

In this chapter, we build up from the simplest idea of fitting a straight line through data to classification, and finally to Bayesian ridge regression.

Fitting a line through data

Now, we get to do some modeling! It's best to start simple; therefore, we'll look at linear regression first.

Linear regression is the first, and therefore, probably the most fundamental model—a straight line through data.

Getting ready

The `boston` dataset is perfect to play around with regression. The `boston` dataset has the median home price of several areas in Boston. It also has other factors that might impact housing prices, for example, crime rate.

First, import the `datasets` model, then we can load the dataset:

```
>>> from sklearn import datasets  
>>> boston = datasets.load_boston()
```

How to do it...

Actually, using linear regression in scikit-learn is quite simple. The API for linear regression is basically the same API you're now familiar with from the previous chapter.

First, import the `LinearRegression` object and create an object:

```
>>> from sklearn.linear_model import  
LinearRegression  
>>> lr = LinearRegression()
```

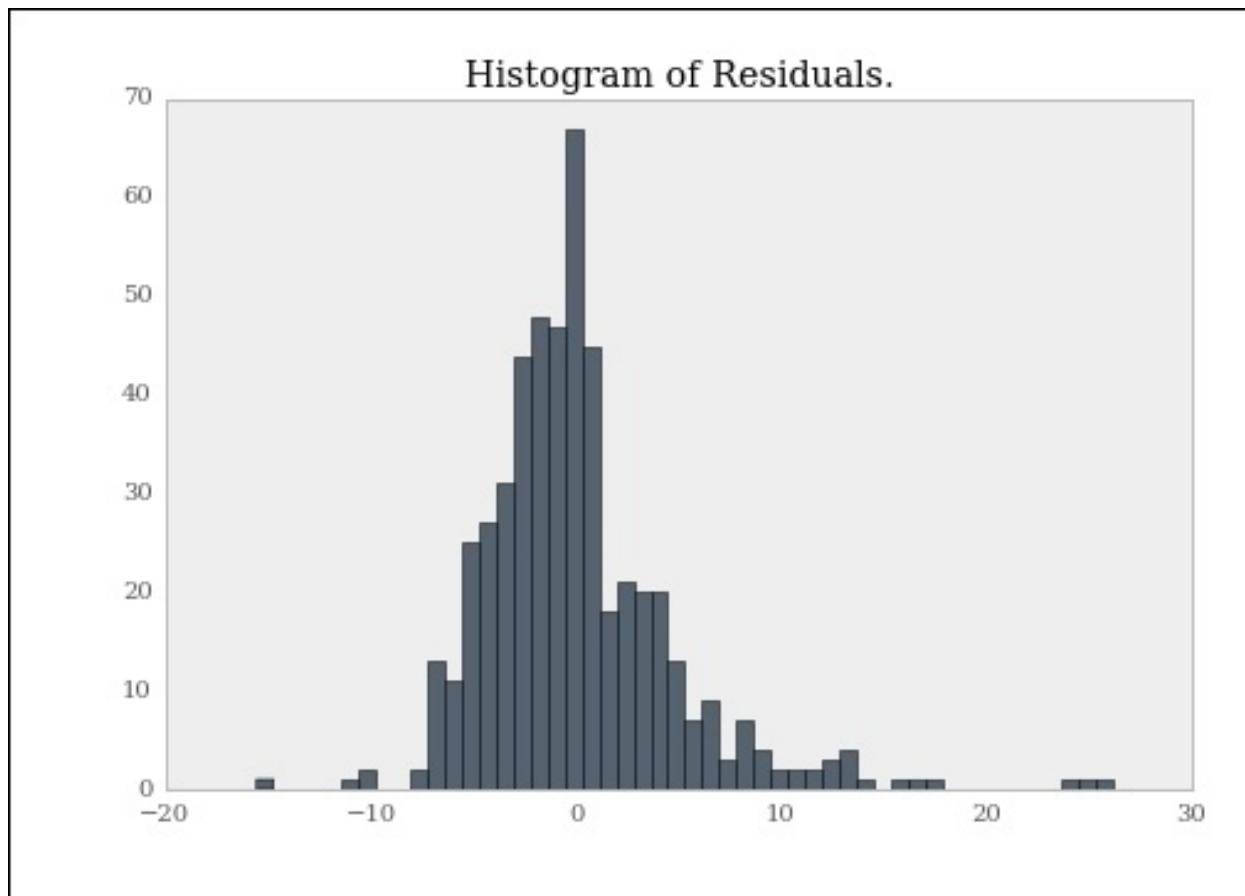
Now, it's as easy as passing the independent and dependent variables to the `fit` method of `LinearRegression`:

```
>>> lr.fit(boston.data, boston.target)  
LinearRegression(copy_X=True,  
fit_intercept=True, normalize=False)
```

Now, to get the predictions, do the following:

```
>>> predictions = lr.predict(boston.data)
```

It's then probably a good idea to look at how close the predictions are to the actual data. We can use a histogram to look at the differences. These are called the **residuals**, as shown:



Let's take a look at the coefficients:

```
>>> lr.coef_
array([-1.07170557e-01,  4.63952195e-02,
       2.08602395e-02,
       2.68856140e+00, -1.77957587e+01,
       3.80475246e+00,
       7.51061703e-04, -1.47575880e+00,
       3.05655038e-01,
      -1.23293463e-02, -9.53463555e-01,
       9.39251272e-03,
      -5.25466633e-01])
```

Tip

A common pattern to express the coefficients of the features and their names is `zip(boston.feature_names, lr.coef_)`.

So, going back to the data, we can see which factors have a negative relationship with the outcome, and also the factors that have a positive relationship. For example, and as expected, an increase in the per capita crime rate by town has a negative relationship with the price of a home in Boston. The per capita crime rate is the first coefficient in the regression.

How it works...

The basic idea of linear regression is to find the set of coefficients of β that satisfy $y = X\beta$, where X is the data matrix. It's unlikely that for the given values of X , we will find a set of coefficients that exactly satisfy the equation; an error term gets added if there is an inexact specification or measurement error. Therefore, the

equation becomes $y = X\beta + \varepsilon$, where ε is assumed to be normally distributed and independent of the X values. Geometrically, this means that the error terms are perpendicular to X . It's beyond the scope of this book, but it might be worth it to prove $E(X\varepsilon) = 0$ to yourself.

In order to find the set of betas that map the X values to y , we minimize the error term. This is done by minimizing the residual sum of squares.

This problem can be solved analytically, with the solution being $\beta = (X^T X)^{-1} X^T \hat{y}$.

There's more...

The `LinearRegression` object can automatically normalize (or scale) the inputs:

```
>>> lr2 = LinearRegression(normalize=True)
>>> lr2.fit(boston.data, boston.target)
LinearRegression(copy_X=True,
fit_intercept=True, normalize=True)
>>> predictions2 = lr2.predict(boston.data)
```

Evaluating the linear regression model

In this recipe, we'll look at how well our regression fits the underlying data. We fit a regression in the last recipe, but didn't pay much attention to how well we actually did it. The first question after we fit the model was clearly "*How well does the model fit?*" In this recipe, we'll examine this question.

Getting ready

Let's use the `lr` object and `boston` dataset—reach back into your code from the *Fitting a line through data* recipe. The `lr` object will have a lot of useful methods now that the model has been fit.

How to do it...

There are some very simple metrics and plots we'll want to look at as well. Let's take another look at the residual plot from the last chapter:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> f = plt.figure(figsize=(7, 5))
>>> ax = f.add_subplot(111)
>>> ax.hist(boston.target - predictions,
bins=50)
>>> ax.set_title("Histogram of Residuals.")
```

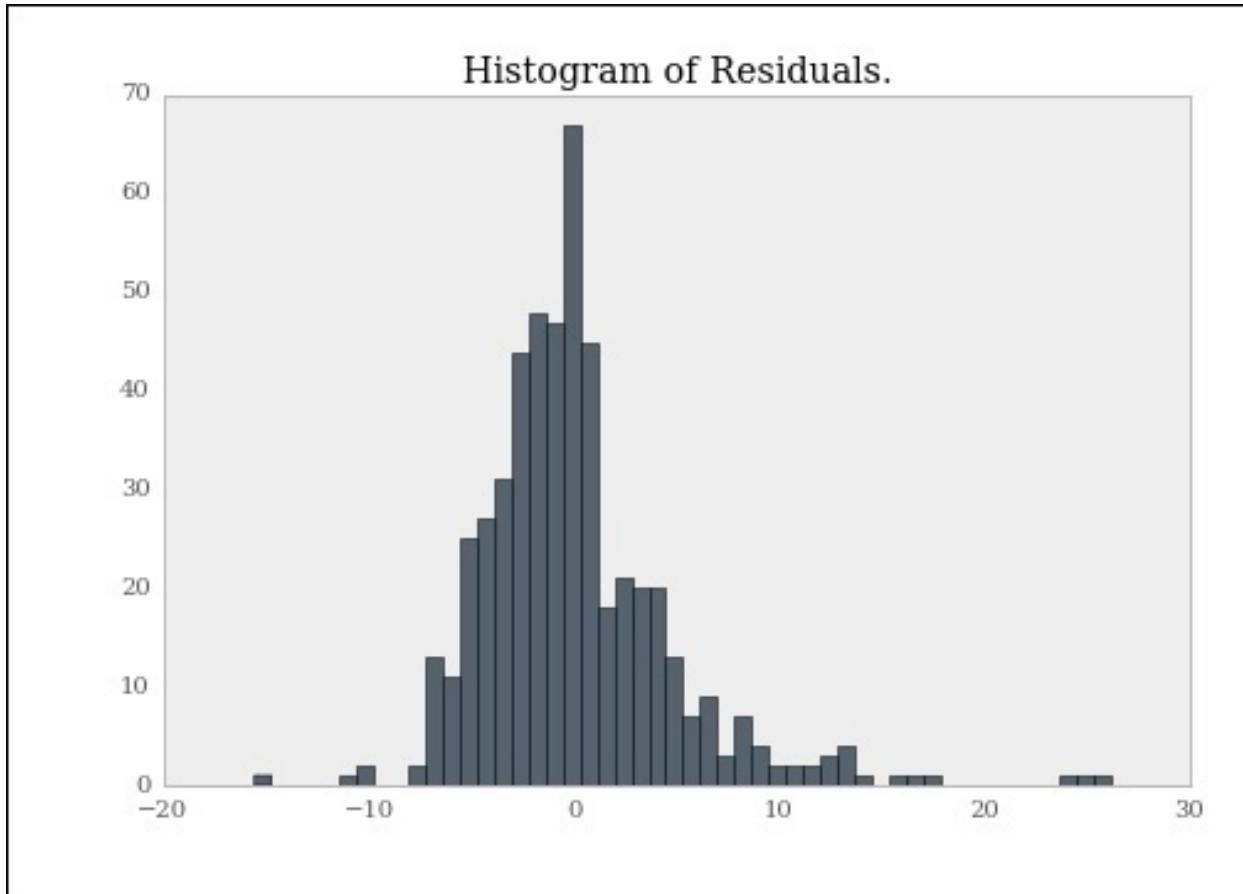
If you're using IPython Notebook, use the `%matplotlib inline` command to render the plots inline. If you're using a regular interpreter, simply type

`f.savefig('myfig.png')` and the plot will be saved for you.

Tip

Plotting is done via `matplotlib`. This isn't the focus of this book, but it's useful to plot your results, so we'll show some basic plotting.

The following is the histogram showing the output:



Like I mentioned previously, the error terms should be normal, with a mean of 0. The residuals are the errors; therefore, this plot should be approximately normal. Visually, it's a good fit, though a bit skewed. We can also look at the mean of the residuals, which should be very close to 0:

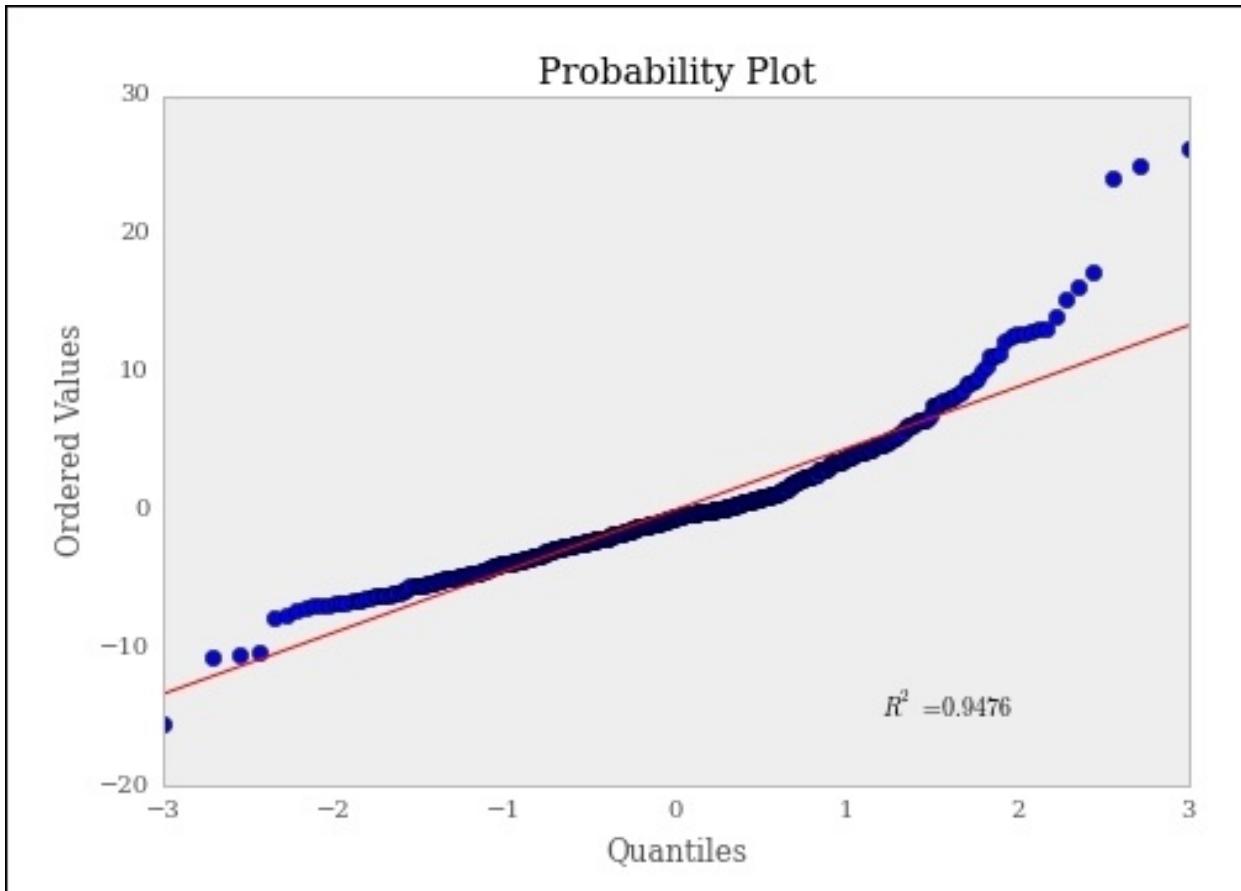
```
>>> np.mean(boston.target - predictions)  
4.3250427394093058e-15
```

Clearly, we are very close.

Another plot worth looking at is a **Q-Q plot**. We'll use **SciPy** here because it has a built-in probability plot:

```
>>> from scipy.stats import probplot  
>>> f = plt.figure(figsize=(7, 5))  
>>> ax = f.add_subplot(111)  
>>> probplot(boston.target - predictions,  
plot=ax)
```

The following screenshot shows the probability plot:



Here, the skewed values we saw earlier are a bit clearer.

We can also look at some other metrics of the fit; **mean squared error (MSE)** and **mean absolute deviation (MAD)** are two common metrics. Let's define each one in Python and use them. Later in the book, we'll look at how scikit-learn has built-in metrics to evaluate the regression models:

```
>>> def MSE(target, predictions):
        squared_deviation = np.power(target -
predictions, 2)
        return np.mean(squared_deviation)
>>> MSE(boston.target, predictions)
21.897779217687496

>>> def MAD(target, predictions):
        absolute_deviation = np.abs(target -
predictions)
        return np.mean(absolute_deviation)

>>> MAD(boston.target, predictions)
3.2729446379969396
```

How it works...

The formula for MSE is very simple:

$$E(\hat{y}_t - y_i)^2$$

It takes each predicted value's deviance from the actual value, squares it, and then averages all the squared terms. This is actually what we optimized to find the best set of coefficients for linear regression. The **Gauss-Markov** theorem actually guarantees that the solution to linear regression is the best in the sense that the coefficients have the smallest expected squared error and are unbiased. In the *Using ridge regression to overcome linear regression's shortfalls* recipe, we'll look at what happens when we're okay with our coefficients being biased.

MAD is the expected error for the absolute errors:

$$E|\hat{y}_t - y_i|$$

MAD isn't used when fitting the linear regression, but it's worth taking a look at. Why? Think about what each one is doing and which errors are more important in each case. For example, with MSE, the larger errors get penalized more than the other terms because of the square term.

There's more...

One thing that's been glossed over a bit is the fact that the coefficients themselves are random variables, and therefore, they have a distribution. Let's use bootstrapping to look at the distribution of the coefficient for the crime rate. Bootstrapping is a very common technique to get an understanding of the uncertainty of an estimate:

```
>>> n_bootstraps = 1000
>>> len_boston = len(boston.target)
>>> subsample_size = np.int(0.5*len_boston)
>>> subsample = lambda:
    np.random.choice(np.arange(0, len_boston),
                     size=subsample_size)

>>> coefs = np.ones(n_bootstraps) #pre-allocate
the space for the coefs

>>> for i in range(n_bootstraps):
    subsample_idx = subsample()
    subsample_X = boston.data[subsample_idx]
    subsample_y =
boston.target[subsample_idx]

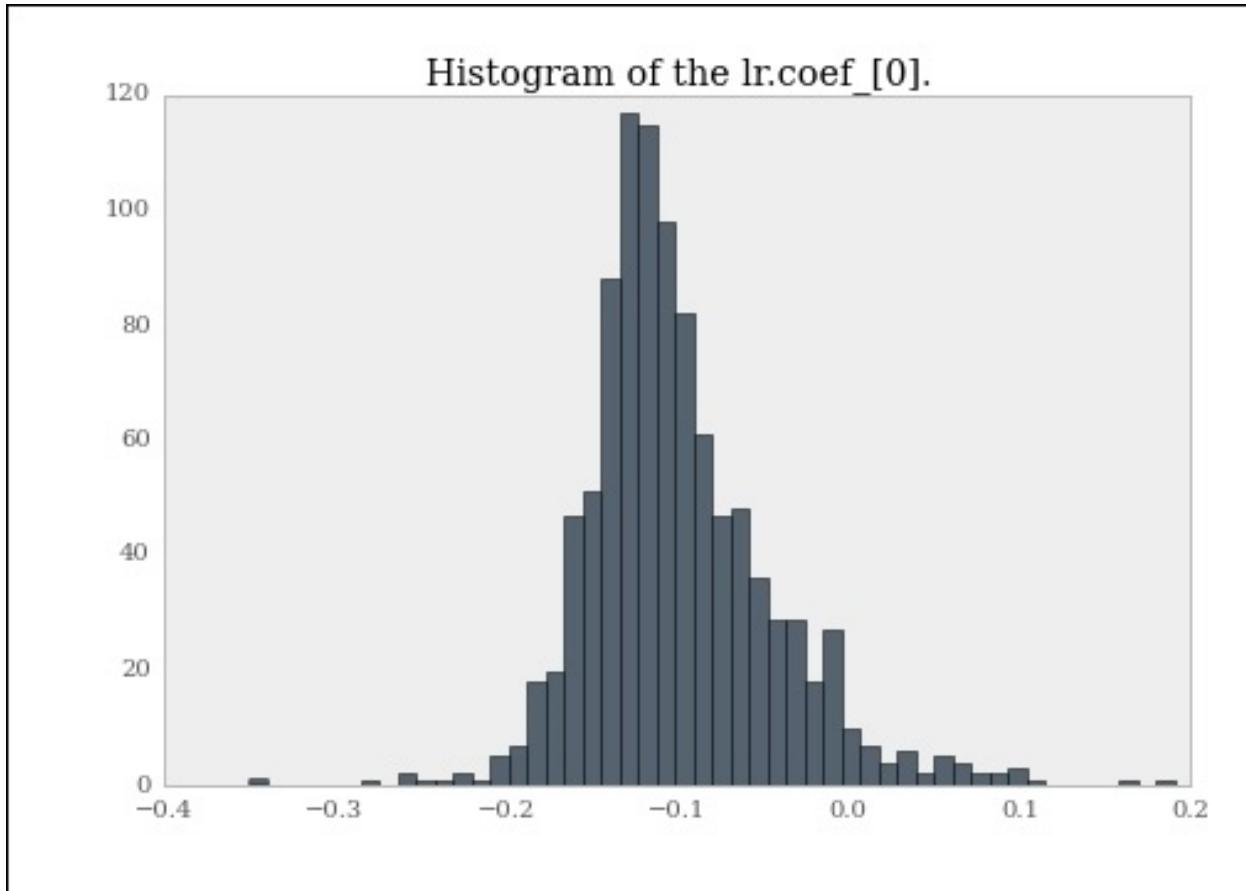
>>> lr.fit(subsample_X, subsample_y)

>>> coefs[i] = lr.coef_[0]
```

Now, we can look at the distribution of the coefficient:

```
>>> import matplotlib.pyplot as plt  
>>> f = plt.figure(figsize=(7, 5))  
>>> ax = f.add_subplot(111)  
>>> ax.hist(coefs, bins=50)  
>>> ax.set_title("Histogram of the  
lr.coef_[0].")
```

The following is the histogram that gets generated:



We might also want to look at the bootstrapped confidence interval:

```
>>> np.percentile(coefs, [2.5, 97.5])
array([-0.18566145,  0.03142513])
```

This is interesting; there's actually reason to believe that the crime rate might not have an impact on the home prices. Notice how zero is within CI, which means that it may not play a role.

It's also worth pointing out that bootstrapping can lead to a potentially better estimation for coefficients because the bootstrapped mean with converge to the true mean is faster than the coefficient found using regular estimation when in the limit.

Using ridge regression to overcome linear regression's shortfalls

In this recipe, we'll learn about ridge regression. It is different from vanilla linear regression; it introduces a regularization parameter to "shrink" the coefficients. This is useful when the dataset has collinear factors.

Getting ready

Let's load a dataset that has a low effective rank and compare ridge regression with linear regression by way of the coefficients. If you're not familiar with rank, it's the smaller of the linearly independent columns and the linearly independent rows. One of the assumptions of linear regression is that the data matrix is of "full rank".

How to do it...

First, use `make_regression` to create a simple dataset with three predictors, but an effective rank of 2. **Effective rank** means that while technically the matrix is of full rank, many of the columns have a high degree of colinearity:

```
>>> from sklearn.datasets import make_regression
>>> reg_data, reg_target =
make_regression(n_samples=2000,
                           n_features=3,
effective_rank=2, noise=10)
```

First, let's take a look at regular linear regression:

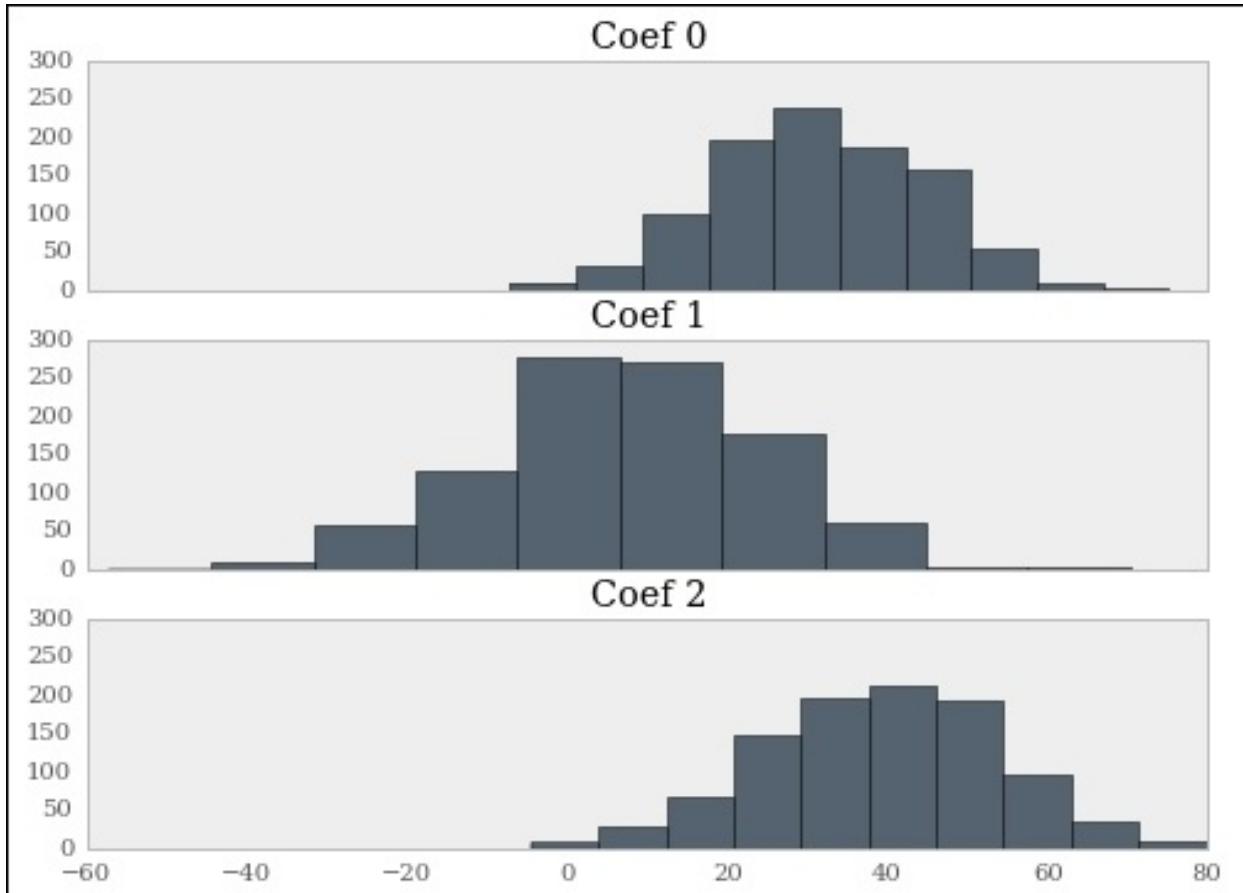
```
>>> import numpy as np
>>> n_bootstraps = 1000
>>> len_data = len(reg_data)
>>> subsample_size = np.int(0.75*len_data)
>>> subsample = lambda:
np.random.choice(np.arange(0, len_data),
                           size=subsample_size)
>>> coefs = np.ones((n_bootstraps, 3))

>>> for i in range(n_bootstraps):
    subsample_idx = subsample()
    subsample_X = reg_data[subsample_idx]
    subsample_y = reg_target[subsample_idx]

>>> lr.fit(subsample_X, subsample_y)
```

```
>>> coefs[i][0] = lr.coef_[0]
>>> coefs[i][1] = lr.coef_[1]
>>> coefs[i][2] = lr.coef_[2]
```

The following is the output that gets generated:



Follow the same procedure with `Ridge`, and have a look at the output:

```
>>> r = Ridge()
>>> n_bootstraps = 1000
```

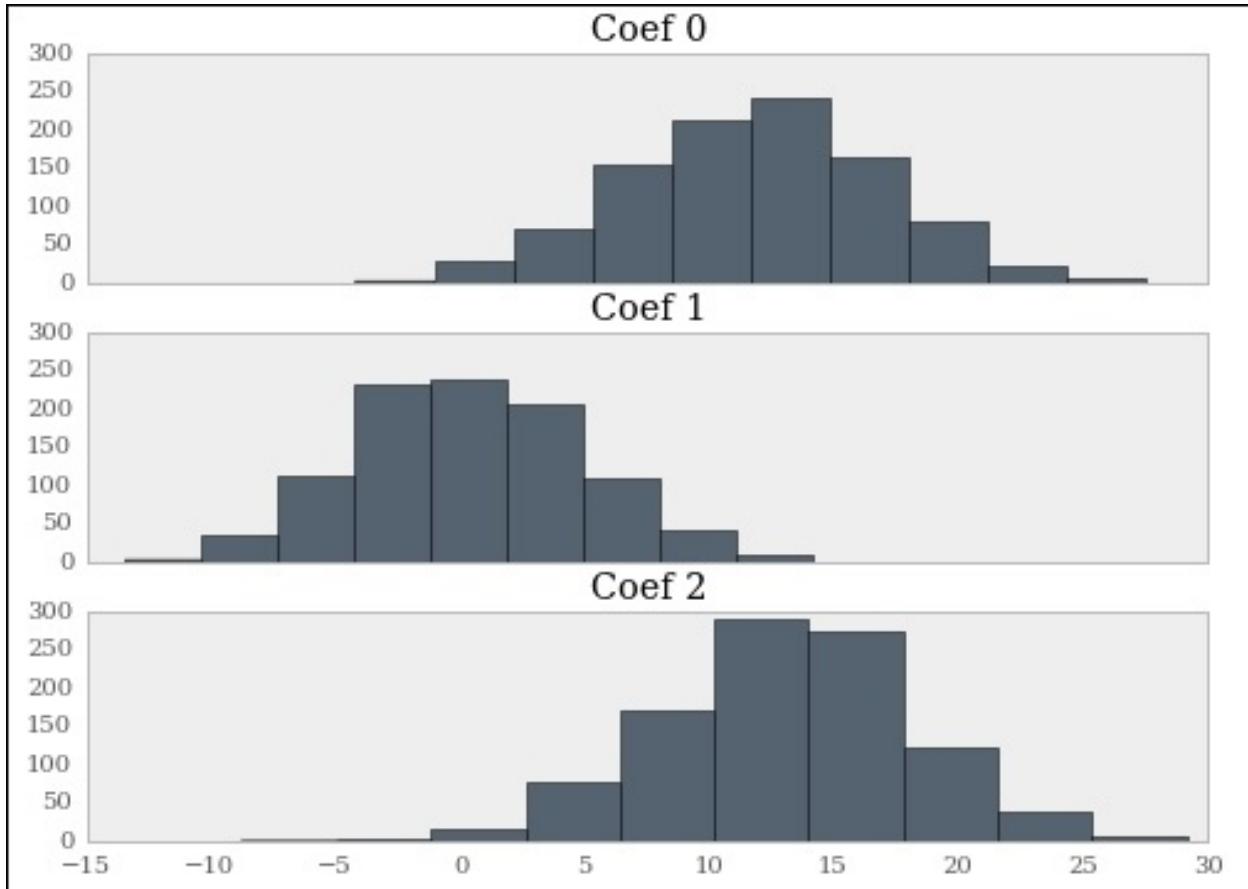
```

>>> len_data = len(reg_data)
>>> subsample_size = np.int(0.75*len_data)
>>> subsample = lambda:
np.random.choice(np.arange(0, len_data),
size=subsample_size)

coefs_r = np.ones((n_bootstraps, 3))
# carry out the same procedure from above

```

The following is the output that gets generated:



Don't let the similar width of the plots fool you; the

coefficients for ridge regression are much closer to 0. Let's look at the average spread between the coefficients:

```
>>> np.mean(coefs - coefs_r, axis=0)
#coefs_r stores the ridge regression
coefficients
array([ 22.19529525,  49.54961002,
8.27708536])
```

So, on an average, the coefficients for linear regression are much higher than the ridge regression coefficients. This difference is the bias in the coefficients (forgetting, for a second, the potential bias of the linear regression coefficients). So then, what is the advantage of ridge regression? Well, let's look at the variance of our coefficients:

```
>>> np.var(coefs, axis=0)
array([ 184.50845658, 150.16268077,
263.39096391])
```

```
>>> np.var(coefs_r, axis=0)
array([ 21.35161646, 23.95273241,
17.34020101])
```

The variance has been dramatically reduced. This is the bias-variance trade-off that is so often discussed in machine learning. The next recipe will introduce how to tune the regularization parameter in ridge regression, which is at the heart of this trade-off.

How it works...

Speaking of the regularization parameter, let's go through how ridge regression differs from linear regression. As was already shown, linear regression works, but it finds the vector of betas that minimize $\|\hat{y} - X\beta\|^2$.

Ridge regression finds the vector of betas that minimize $\|\hat{y} - X\beta\|^2 + \|\Gamma X\|^2$.

Γ is typically *all*, or it's some scalar times the identity matrix. We actually used the default alpha when initializing ridge regression.

Now that we created the object, we can look at its attributes:

```
>>> r #notice the alpha parameter
Ridge(alpha=1.0, copy_X=True,
      fit_intercept=True, max_iter=None,
      normalize=False, solver='auto', tol=0.001)
```

This minimization has the following solution:

$$\beta = (X^T X + \Gamma^T \Gamma)^{-1} X y$$

The previous solution is the same as linear regression, except for the $\Gamma^T \Gamma$ term. For a matrix A , AA^T is symmetric, and thus positive semidefinite. So, thinking about the translation of matrix algebra from scalar algebra, we effectively divide by a larger number.

Multiplication by an inverse is analogous to division. So, this is what squeezes the coefficients towards 0. This is a bit of a crude explanation; for a deeper understanding, you should look at the connections between SVD and ridge regression.

Optimizing the ridge regression parameter

Once you start using ridge regression to make predictions or learn about relationships in the system you're modeling, you'll start thinking about the choice of alpha.

For example, using OLS regression might show some relationship between two variables; however, when regularized by some alpha, the relationship is no longer significant. This can be a matter of whether a decision needs to be taken.

Getting ready

This is the first recipe where we'll tune the parameters for a model. This is typically done by cross-validation. There will be recipes laying out a more general way to do this in later recipes, but here we'll walkthrough to be able to tune ridge regression.

If you remember, in ridge regression, the gamma parameter is typically represented as alpha in scikit-learn when calling `RidgeRegression`; so, the question that arises is what the best alpha is. Create a regression dataset, and then let's get started:

```
>>> from sklearn.datasets import make_regression  
>>> reg_data, reg_target =  
make_regression(n_samples=100,  
                n_features=2,  
effective_rank=1, noise=10)
```

How to do it...

In the `linear_models` module, there is an object called `RidgeCV`, which stands for **ridge cross-validation**. This performs a cross-validation similar to **leave-one-out cross-validation (LOOCV)**.

Under the hood, it's going to train the model for all samples except one. It'll then evaluate the error in predicting this one test case:

```
>>> from sklearn.linear_model import RidgeCV
>>> rcv = RidgeCV(alphas=np.array([.1, .2, .3,
... .4]))
>>> rcv.fit(reg_data, reg_target)
RidgeCV(alphas=array([ 0.1,  0.2,  0.3,  0.4]), cv=None,
        fit_intercept=True, gcv_mode=None,
        loss_func=None,
        normalize=False, score_func=None,
        scoring=None,
        store_cv_values=False)
```

After we fit the regression, the `alpha` attribute will be the *best* alpha choice:

```
>>> rcv.alpha_
0.1000000000000001
```

In the previous example, it was the first choice. We might

want to hone in on something around .1:

```
>>> rcv2 = RidgeCV(alphas=np.array([.08, .09,
... .1, .11, .12]))
>>> rcv2.fit(reg_data, reg_target)
RidgeCV(alphas=array([ 0.08,  0.09,  0.1 ,
0.11,  0.12]), cv=None,
         fit_intercept=True,
gcv_mode=None,
         loss_func=None,
normalize=False,
         score_func=None,
scoring=None,
         store_cv_values=False)

>>> rcv2.alpha_
0.08
```

We can continue this hunt, but hopefully, the mechanics are clear.

How it works...

The mechanics might be clear, but we should talk a little more about the why and define what was meant by "best". At each step in the cross-validation process, the model scores an error against the test sample. By default, it's essentially a squared error. Check out the *There's more...* section for more details.

We can force the `RidgeCV` object to store the cross-validation values; this will let us visualize what it's doing:

```
>>> alphas_to_test = np.linspace(0.01, 1)
>>> rcv3 = RidgeCV(alphas=alphas_to_test,
store_cv_values=True)
>>> rcv3.fit(reg_data, reg_target)
```

As you can see, we test a bunch of points (50 in total) between 0.01 and 1. Since we passed `store_cv_values` as true, we can access these values:

```
>>> rcv3.cv_values_.shape
(100, 50)
```

So, we had 100 values in the initial regression and tested 50 different alpha values. We now have access to the errors of all 50 values. So, we can now find the smallest mean error and choose it as alpha:

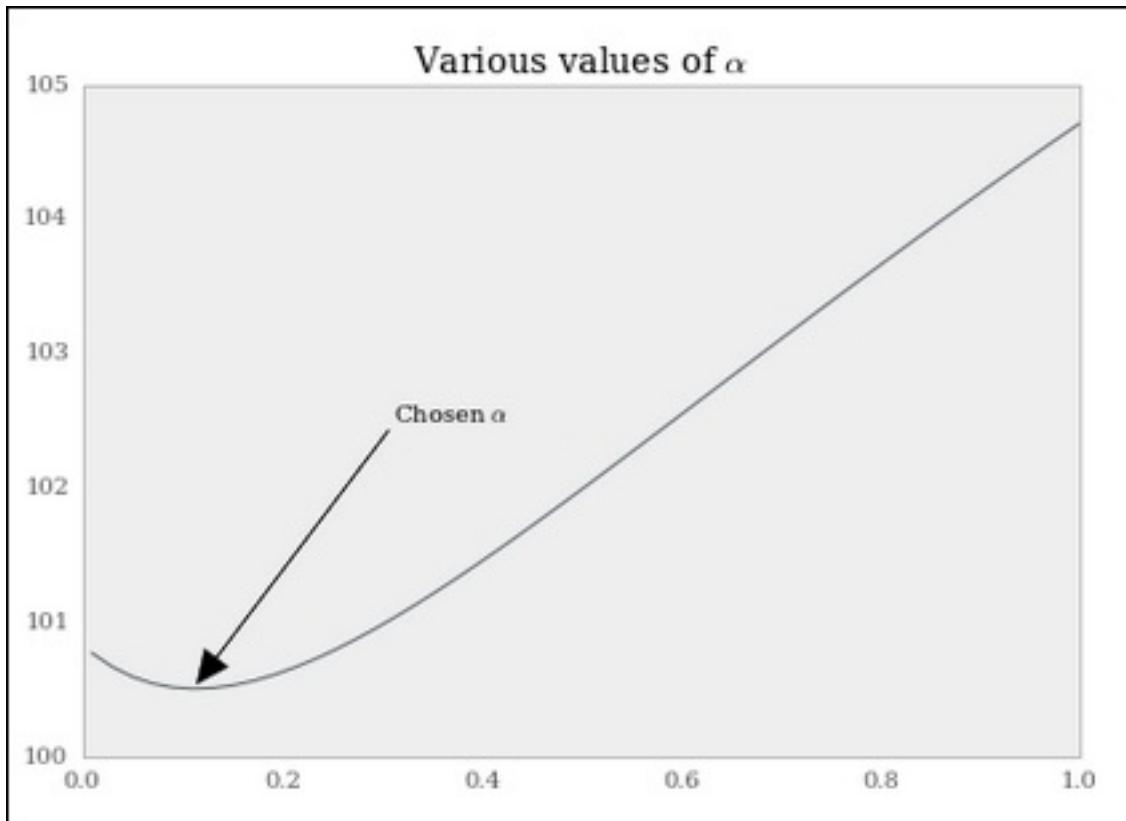
```
>>> smallest_idx =  
rcv3.cv_values_.mean(axis=0).argmin()  
>>> alphas_to_test[smallest_idx]
```

The question that arises is "*Does RidgeCV agree with our choice?*" Use the following command to find out:

```
>>> rcv3.alpha_  
0.01
```

Beautiful!

It's also worthwhile to visualize what's going on. In order to do that, we'll plot the mean for all 50 test alphas.



There's more...

If we want to use our own scoring function, we can do that as well. Since we looked up MAD before, let's use it to score the differences. First, we need to define our loss function:

```
>>> def MAD(target, predictions):
    absolute_deviation = np.abs(target -
predictions)
    return absolute_deviation.mean()
```

After we define the loss function, we can employ the `make_scoring` function in `sklearn`. This will take care of standardizing our function so that scikit's objects know how to use it. Also, because this is a loss function and not a score function, the lower the better, and thus the need to let `sklearn` to flip the sign to turn this from a maximization problem into a minimization problem:

```
>>> import sklearn
>>> MAD = sklearn.metrics.make_scoring(MAD,
greater_is_better=False)
>>> rcv4 = RidgeCV(alphas=alphas_to_test,
store_cv_values=True,
           scoring=MAD)
>>> rcv4.fit(reg_data, reg_target)
>>> smallest_idx =
rcv4.cv_values_.mean(axis=0).argmin()
>>> alphas_to_test[smallest_idx]
```

0 . 2322

Using sparsity to regularize models

The **least absolute shrinkage and selection operator (LASSO)** method is very similar to ridge regression and LARS. It's similar to Ridge Regression in the sense that we penalize our regression by some amount, and it's similar to LARS in that it can be used as a parameter selection, and it typically leads to a sparse vector of coefficients.

Getting ready

To be clear, lasso regression is not a panacea. There can be computation consequences to using lasso regression. As we'll see in this recipe, we'll use a loss function that isn't differential, and therefore, requires special, and more importantly, performance-impairing workarounds.

How to do it...

Let's go back to the trusty `make_regression` function and create a dataset with the same parameters:

```
>>> from sklearn.datasets import make_regression  
>>> reg_data, reg_target =  
make_regression(n_samples=200, n_features=500,  
                n_informative=5,  
                noise=5)
```

Next, we need to import the `Lasso` object:

```
>>> from sklearn.linear_model import Lasso  
>>> lasso = Lasso()
```

`Lasso` contains many parameters, but the most interesting parameter is `alpha`. It scales the penalization term of the `Lasso` method, which we'll look at in the *How it works...* section. For now, leave it as `1`. As an aside, and much like ridge regression, if this term is `0`, `lasso` is equivalent to linear regression:

```
>>> lasso.fit(reg_data, reg_target)
```

Again, let's see how many of the coefficients remain nonzero:

```
>>> np.sum(lasso.coef_ != 0)
```

```
>>> lasso_0 = Lasso(0)
>>> lasso_0.fit(reg_data, reg_target)
>>> np.sum(lasso_0.coef_ != 0)
500
```

None of our coefficients turn out to be 0, which is what we expect. Actually, if you run this, you might get a warning from scikit-learn that advises you to choose LinearRegression.

How it works...

For linear regression, we minimized the squared error. Here, we're still going to minimize the squared error, but we'll add a penalization term that will induce the scarcity. The equation looks like the following:

$$\sum e_i + \lambda \|\beta\|_1$$

An alternate way of looking at this is to minimize the residual sum of squares:

$$\text{RSS}(\beta) \text{ such that } \|\beta\|_1 < \beta$$

This constraint is what leads to the scarcity. Lasso regression's constraint creates a hypercube around the origin (the coefficients being the axis), which means that the most extreme points are the corners, where many of the coefficients are 0. Ridge regression creates a hypersphere due to the constraint of the l2 norm being less than some constant, but it's very likely that coefficients will not be zero even if they are constrained.

Lasso cross-validation

Choosing the most appropriate lambda is a critical problem. We can specify the lambda ourselves or use cross-validation to find the best choice given the data at hand:

```
>>> from sklearn.linear_model import LassoCV  
>>> lassocv = LassoCV()  
>>> lassocv.fit(reg_data, reg_target)
```

`lassocv` will have, as an attribute, the most appropriate lambda. scikit-learn mostly uses alpha in its notation, but the literature uses lambda:

```
>>> lassocv.alpha_  
0.80722126078646139
```

The number of coefficients can be accessed in the regular manner:

```
>>> lassocv.coef_[:5]  
array([0., 42.41, 0., 0., -0.])
```

Letting `lassocv` choose the appropriate best fit leaves us with 11 nonzero coefficients:

```
>>> np.sum(lassocv.coef_ != 0)  
11
```

Lasso for feature selection

Lasso can often be used for feature selection for other methods. For example, you might run lasso regression to get the appropriate number of features, and then use these features in another algorithm.

To get the features we want, create a masking array based on the columns that aren't zero, and then filter to keep the features we want:

```
>>> mask = lassocv.coef_ != 0
>>> new_reg_data = reg_data[:, mask]
>>> new_reg_data.shape
(200, 11)
```

Taking a more fundamental approach to regularization with LARS

To borrow from Gilbert Strang's evaluation of the Gaussian elimination, LARS is an idea you probably would've considered eventually had it not been discovered previously by Efron, Hastie, Johnstone, and Tibshirani in their works[1].

Getting ready

Least-angle regression (LARS) is a regression technique that is well suited for high-dimensional problems, that is, $p \gg n$, where p denotes the columns or features and n is the number of samples.

How to do it...

First, import the necessary objects. The data we use will have 200 data points and 500 features. We'll also choose a low noise and a small number of informative features:

```
>>> from sklearn.datasets import make_regression  
>>> reg_data, reg_target =  
make_regression(n_samples=200,  
                           n_features=500,  
n_informative=10, noise=2)
```

Since we used 10 informative features, let's also specify that we want 10 nonzero coefficients in LARS. We will probably not know the exact number of informative features beforehand, but it's useful for learning purposes:

```
>>> from sklearn.linear_model import Lars  
>>> lars = Lars(n_nonzero_coefs=10)  
>>> lars.fit(reg_data, reg_target)
```

We can then verify that LARS returns the correct number of nonzero coefficients:

```
>>> np.sum(lars.coef_ != 0)  
10
```

The question then is why it is more useful to use a smaller number of features. To illustrate this, let's hold out half of the data and train two LARS models, one with 12 nonzero

coefficients and another with no predetermined amount. We use 12 here because we might have an idea of the number of important features, but we might not be sure of the exact number:

```
>>> train_n = 100
>>> lars_12 = Lars(n_nonzero_coefs=12)
>>> lars_12.fit(reg_data[:train_n],
    reg_target[:train_n])

>>> lars_500 = Lars() # it's 500 by default
>>> lars_500.fit(reg_data[:train_n],
    reg_target[:train_n]);
```

Now, to see how well each feature fit the unknown data, do the following:

```
>>> np.mean(np.power(reg_target[train_n:] -
lars_12.predict(reg_data
    [train_n:]), 2))
31.527714163321001
>>> np.mean(np.power(reg_target[train_n:] -
lars_500.predict(reg_data
    [train_n:]), 2))
9.6198147535136237e+30
```

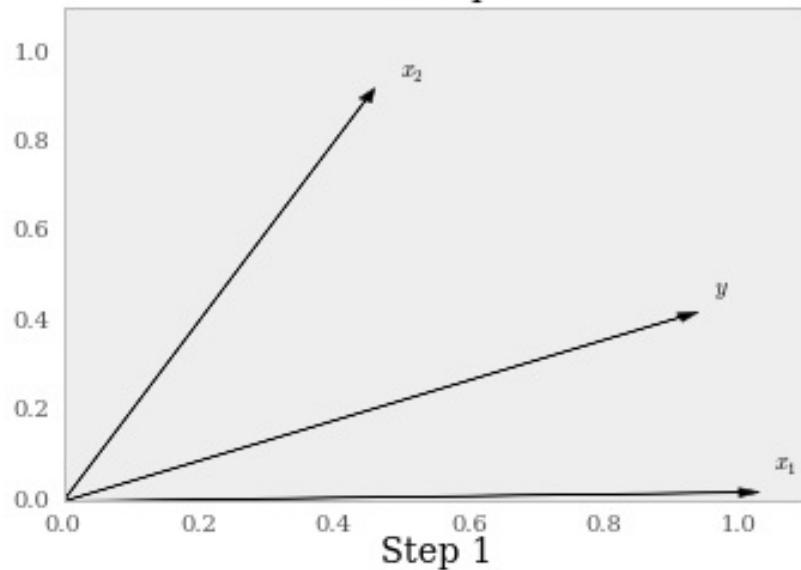
Look again if you missed it; the error on the test set was clearly very high. Herein lies the problem with high-dimensional datasets; given a large number of features, it's typically not too difficult to get a model of good fit on the train sample, but overfitting becomes a huge problem.

How it works...

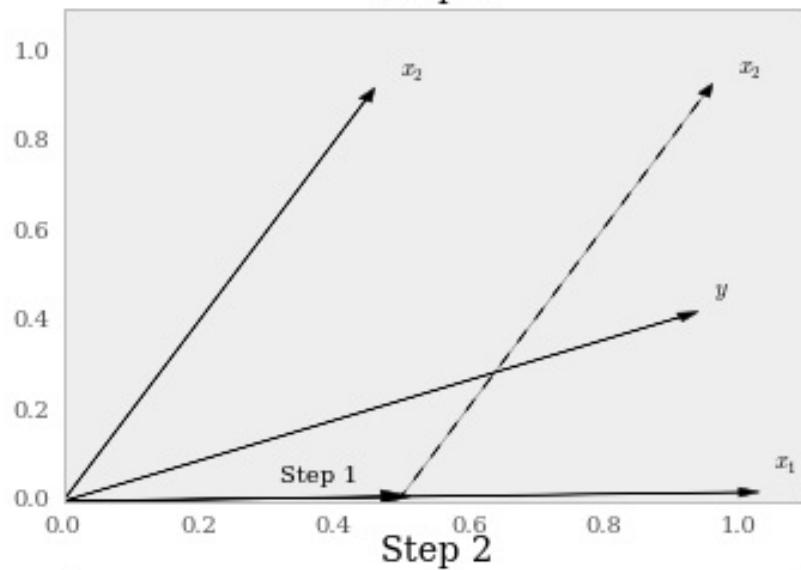
LARS works by iteratively choosing features that are correlated with the residuals. Geometrically, correlation is effectively the least angle between the feature and the residuals; this is how LARS gets its name.

After choosing the first feature, LARS will continue to move in the least angle direction, until a different feature has the same amount of correlation with the residuals. Then, LARS will begin to move in the combined direction of both features. To visualize this, consider the following graph:

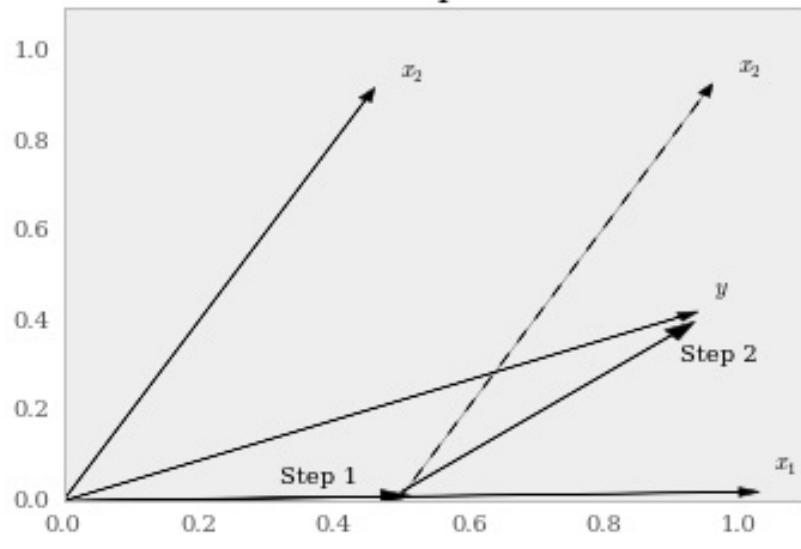
No steps



Step 1



Step 2



So, we move along \mathbf{x}_1 until we get to the point where the *pull* on \mathbf{x}_1 by \mathbf{y} is the same as the *pull* on \mathbf{x}_2 by \mathbf{y} . When this occurs, we move along the path that is equal to the angle between \mathbf{x}_1 and \mathbf{x}_2 divided by 2.

There's more...

Much in the same way we used cross-validation to tune ridge regression, we can do the same with LARS:

```
>>> from sklearn.linear_model import LarsCV  
>>> lcv = LarsCV()  
>>> lcv.fit(reg_data, reg_target)
```

Using cross-validation will help us determine the best number of nonzero coefficients to use. Here, it turns out to be as shown:

```
>>> np.sum(lcv.coef_ != 0)  
44
```

[1]: Efron, Bradley; Hastie, Trevor; Johnstone, Iain and Tibshirani, Robert (2004). "Least Angle Regression". *Annals of Statistics* 32(2): pp. 407–499. doi:10.1214/009053604000000067. MR 2060166.

Using linear methods for classification – logistic regression

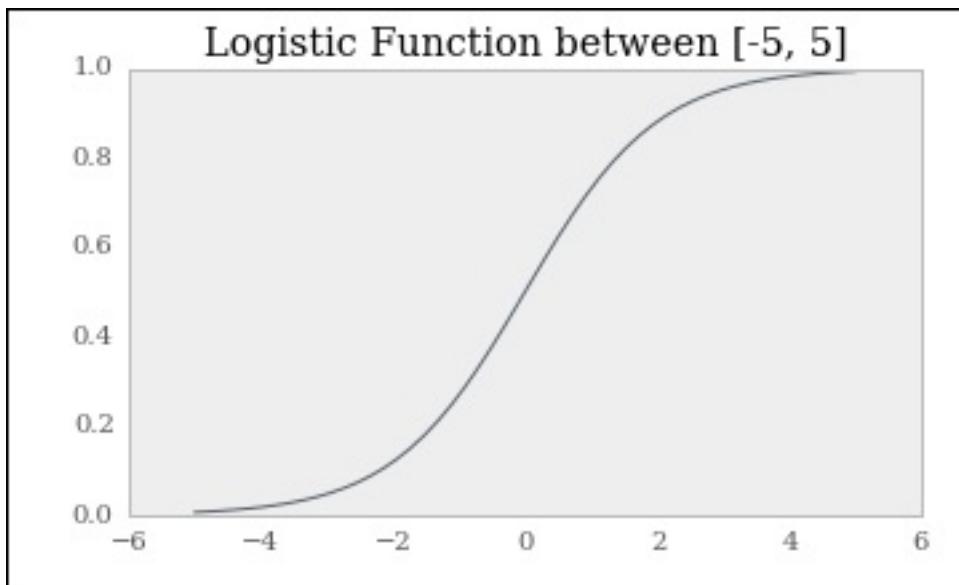
Linear models can actually be used for classification tasks. This involves fitting a linear model to the probability of a certain class, and then using a function to create a threshold at which we specify the outcome of one of the classes.

Getting ready

The function used here is typically the logistic function (surprise!). It's a pretty simple function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Visually, it looks like the following:



Let's use the `make_classification` method, create a dataset, and get to classifying:

```
>>> from sklearn.datasets import  
make_classification  
>>> X, y = make_classification(n_samples=1000,  
n_features=4)
```

How to do it...

The `LogisticRegression` object works in the same way as the other linear models:

```
>>> from sklearn.linear_model import  
LogisticRegression  
>>> lr = LogisticRegression()
```

Since we're good data scientists, we will pull out the last 200 samples to test the trained model on. Since this is a random dataset, it's fine to hold out the last 200; if you're dealing with structured data, don't do this (for example, if you deal with time series data):

```
>>> x_train = x[:-200]  
>>> x_test = x[-200:]  
>>> y_train = y[:-200]  
>>> y_test = y[-200:]
```

We'll discuss more on cross-validation later in the book. Now, we need to fit the model with logistic regression. We'll keep around the predictions on the train set, just like the test set. It's a good idea to see how often you are correct on both sets. Often, you'll be better on the train set; it's a matter of how much worse you are on the test set:

```
>>> lr.fit(x_train, y_train)
```

```
>>> y_train_predictions = lr.predict(x_train)
>>> y_test_predictions = lr.predict(x_test)
```

Now that we have the predictions, let's take a look at how good our predictions were. Here, we'll simply look at the number of times we were correct; later, we'll talk about evaluating classification models in more detail.

The calculation is simple; it's the number of times we were correct over the total sample:

```
>>> (y_train_predictions ==
y_train).sum().astype(float) /
y_train.shape[0]
0.8662499
```

And now the test sample:

```
>>> (y_test_predictions ==
y_test).sum().astype(float) /
y_test.shape[0]
0.900000
```

So, here we were correct about as often in the test set as we were in the train set. Sadly, in practice, this isn't often the case.

The question then changes to how to move on from the logistic function to a method by which we can classify groups.

First, recall the linear regression hopes offending the linear function that fits the expected value of Y , given the values of X ; this is $E(Y|X) = X\beta$. Here, the Y values are the probabilities of the classes. Therefore, the problem we're trying to solve is $E(p|X) = X\beta$. Then, once the threshold is applied, this becomes $\text{Logit}(p) = X\beta$. The idea expanded is how other forms of regression work, for example, Poisson.

There's more...

You'll surely see this again. There will be a situation where one class is weighted differently from the other classes; for example, one class may be 99 percent of cases. This situation will pop up all over the place in the classification work. The canonical example is fraud detection, where most transactions aren't fraud, but the cost associated with misclassification is asymmetric between classes.

Let's create a classification problem with 95 percent imbalance and see how the basic stock logistic regression handles this case:

```
>>> x, y = make_classification(n_samples=5000,  
n_features=4,  
weights=[.95])  
  
>>> sum(y) / (len(y)*1.) #to confirm the class  
imbalance  
0.0555
```

Create the train and test sets, and then fit logistic regression:

```
>>> x_train = x[:-500]  
>>> x_test = x[-500:]  
>>> y_train = y[:-500]
```

```
>>> y_test = y[-500:]

>>> lr.fit(X_train, y_train)
>>> y_train_predictions = lr.predict(X_train)
>>> y_test_predictions = lr.predict(X_test)
```

Now, to see how well our model fits the data, do the following:

```
>>> (y_train_predictions ==
y_train).sum().astype(float) /
    y_train.shape[0]
>>> 0.96977

>>> (y_test_predictions ==
y_test).sum().astype(float) / y_test.shape[0]
>>> 0.97999
```

At first, it looks like we did well, but it turns out that when we always guessed that a transaction was not fraud (or class 0 in general) we were right around 95 percent of the time. If we look at how well we did in classifying the 1 class, it's not nearly as good:

```
>>> (y_test[y_test==1] ==
y_test_predictions[y_test==1])
    .sum().astype(float) /
y_test[y_test==1].shape[0]
0.583333
```

Hypothetically, we might care more about identifying

fraud cases than non-fraud cases; this could be due to a business rule, so we might alter how we weigh the correct and incorrect values.

By default, the classes are weighted (and thus resampled) in accordance with the inverse of the class weights of the training set. However, because we care more about fraud cases, let's oversample the fraud relative to nonfraud cases.

We know that our relative weighting right now is 95 percent nonfraud; let's change this to overweight fraud cases:

```
>>> lr = LogisticRegression(class_weight={0: .15, 1: .85})  
>>> lr.fit(X_train, y_train)
```

Let's predict the outputs again:

```
>>> y_train_predictions = lr.predict(X_train)  
>>> y_test_predictions = lr.predict(X_test)
```

We can see that we did a much better job on classifying the fraud cases:

```
>>> (y_test[y_test==1] ==  
y_test_predictions[y_test==1]).sum().astype(float) / y_test[y_test==1].shape[0]  
0.875
```

But, at what expense do we do this? To find out, use the following command:

```
>>> (y_test_predictions ==  
y_test).sum().astype(float) / y_test.shape[0]  
0.967999
```

Here, there's only about 1 percent less accuracy. Whether that's okay depends on your problem. Put in the context of the problem, if the estimated cost associated with fraud is sufficiently large, it can eclipse the cost associated with tracking fraud.

Directly applying Bayesian ridge regression

In the *Using ridge regression to overcome linear regression's shortfalls* recipe, we discussed the connections between the constraints imposed by ridge regression from an optimization standpoint. We also discussed the Bayesian interpretation of priors on the coefficients, which attract the mass of the density towards the prior, which often has a mean of 0.

So, now we'll look at how we can directly apply this interpretation through scikit-learn.

Getting ready

Ridge and lasso regression can both be understood through a Bayesian lens as opposed to an optimization lens. Only Bayesian ridge regression is implemented by scikit-learn, but in the *How it works...* section, we'll look at both cases.

First, as usual, let's create some regression data:

```
>>> from sklearn.datasets import make_regression  
>>> X, y = make_regression(1000, 10,  
n_informative=2, noise=20)
```

How to do it...

We can just "throw" ridge regression at the problem with a few simple steps:

```
>>> from sklearn.linear_model import  
BayesianRidge  
  
>>> br = BayesianRidge()
```

The two sets of coefficients of interest are

α_1/α_2 and λ_1/λ_2 . The alphas are the hyperparameters for the prior over the alpha parameter, and the lambda are the hyperparameters of the prior over the lambda parameter.

First, let's fit a model without any modification to the hyperparameters:

```
>>> br.fit(x, y)  
>>> br.coef_  
array([0.3000136 , -0.33023408,  68.166673,  
-0.63228159,  0.07350987,  
-0.90736606,  0.38851709, -0.8085291 ,  
0.97259451,  68.73538646])
```

Now, if we modify the hyperparameters, notice the slight changes in the coefficients:

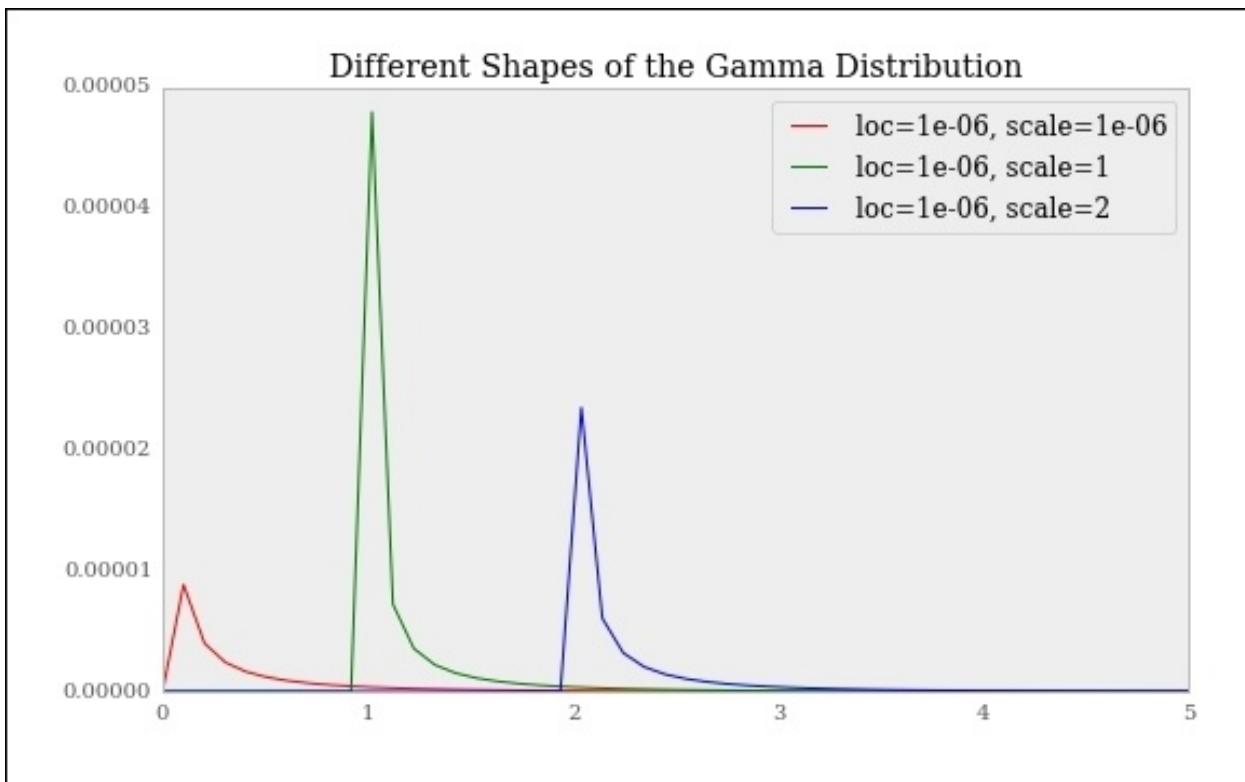
```
>>> br_alpha = BayesianRidge(alpha_1=10,
```

```
lambda_1=10)
>>> br_alphas.fit(X, y)
>>> br_alphas.coef_
array([0.30054387, -0.33130025, 68.10432626,
-0.63056712,
       0.07751436, -0.90919326, 0.39020878,
-0.80822013,
       0.97497567, 68.67409658])
```

How it works...

For Bayesian ridge regression, we assume a prior over the errors and alpha. Both these priors are gamma distributions.

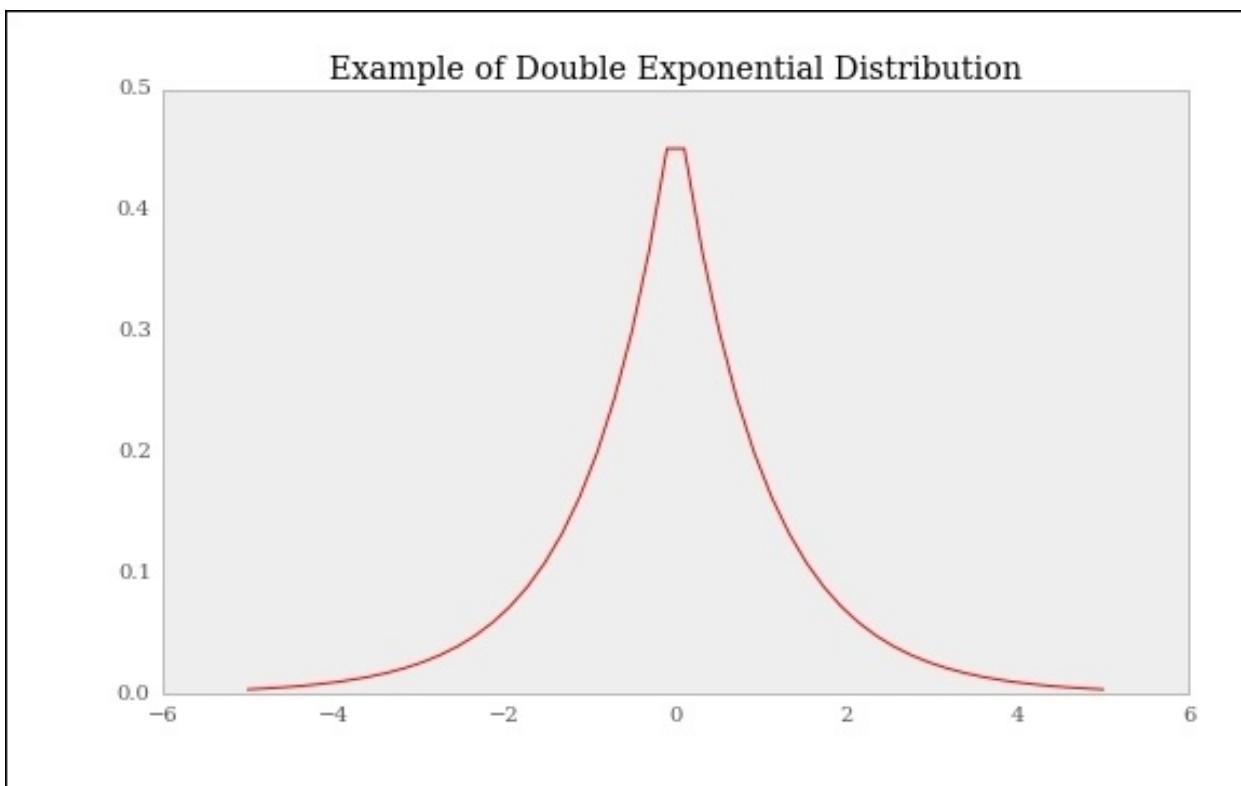
The gamma distribution is a very flexible distribution. Here are some of the different shapes the gamma distribution can take given the different parameterization techniques for location and scale. **1e-06** is the default parameterization of `BayesianRidge` in scikit-learn:



As you can see, the coefficients are naturally shrunk towards 0, especially with a very small location parameter.

There's more...

Like I mentioned earlier, there's also a Bayesian interpretation of lasso regression. Imagine we set priors over the coefficients; remember that they are random numbers themselves. For lasso regression, we will choose a prior that naturally produces 0s, for example, the double exponential.



Notice the peak around **0**. This will naturally lead to the zero coefficients in lasso regression. By tuning the

hyperparameters, it's also possible to create 0 coefficients that more or less depend on the setup of the problem.

Using boosting to learn from errors

Gradient boosting regression is a technique that learns from its mistakes. Essentially, it tries to fit a bunch of weak learners. There are two things to note:

- Individually, each learner has poor accuracy, but together they can have very good accuracy
- They're applied sequentially, which means that each learner becomes an expert in the mistakes of the prior learner

Getting ready

Let's use some basic regression data and see how **gradient boosting** regression (henceforth, GBR) works:

```
>>> from sklearn.datasets import make_regression  
>>> X, y = make_regression(1000, 2, noise=10)
```

How to do it...

GBR is part of the ensemble module because it's an ensemble learner. This is the name for the idea behind using many weak learners to simulate a strong learner:

```
>>> from sklearn.ensemble import  
GradientBoostingRegressor as GBR  
>>> gbr = GBR()  
>>> gbr.fit(X, y)  
>>> gbr_preds = gbr.predict(X)
```

Clearly, there's more to fitting a usable model, but this pattern should be pretty clear by now.

Now, let's fit a basic regression as well so that we can use it as the baseline:

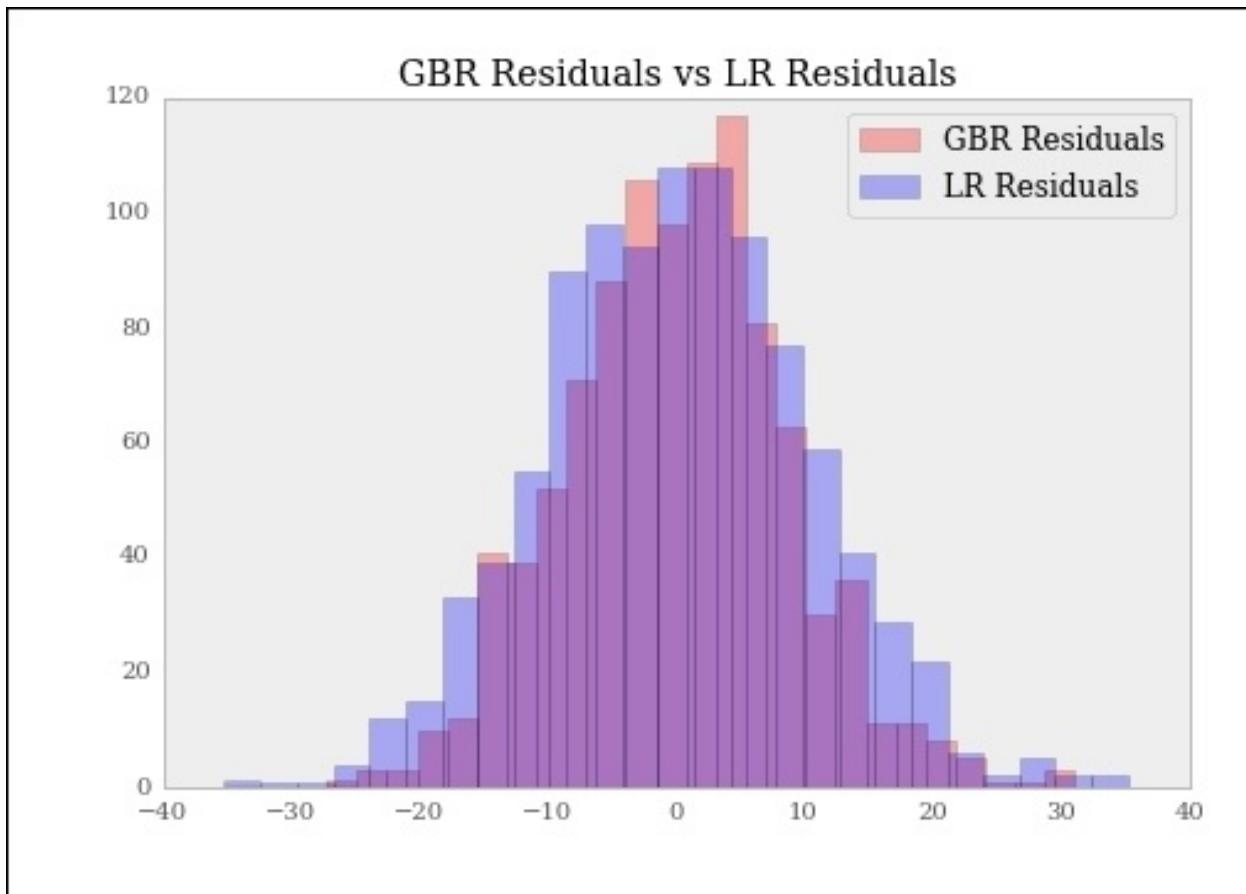
```
>>> from sklearn.linear_model import  
LinearRegression  
>>> lr = LinearRegression()  
>>> lr.fit(X, y)  
>>> lr_preds = lr.predict(X)
```

Now that we have a baseline, let's see how well GBR performed against linear regression.

I'll leave it as an exercise for you to plot the residuals, but to get started, do the following:

```
>>> gbr_residuals = y - gbr_preds  
>>> lr_residuals = y - lr_preds
```

The following will be the output:



It looks like GBR has a better fit, but it's a bit hard to tell.
Let's take the 95 percent CI and compare:

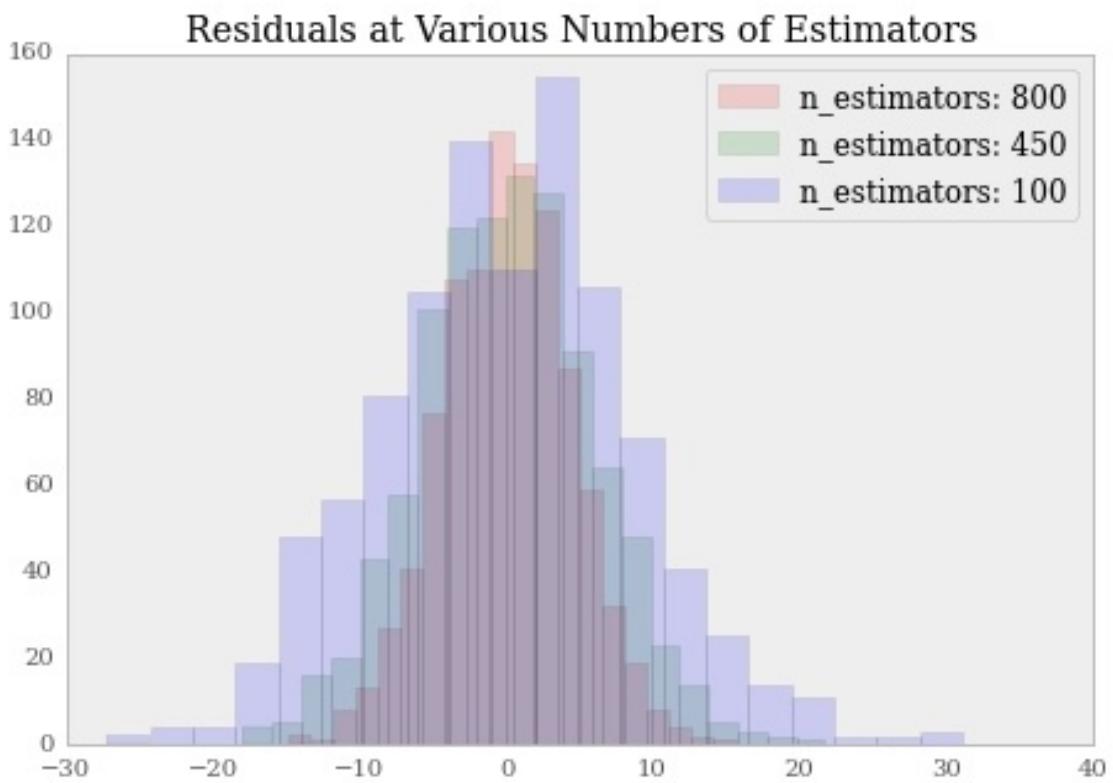
```
>>> np.percentile(gbr_residuals, [2.5, 97.5])  
array([-16.05443674, 17.53946294])  
  
>>> np.percentile(lr_residuals, [2.5, 97.5])
```

```
array([-20.05434912,  19.80272884])
```

So, GBR clearly fits a bit better; we can also make several modifications to the GBR algorithm, which might improve performance. I'll show an example here, then we'll walkthrough the different options in the *How it works...* section:

```
>>> n_estimators = np.arange(100, 1100, 350)
>>> gbrs = [GBR(n_estimators=n_estimator) for
n_estimator in
            n_estimators]
>>> residuals = {}
>>> for i, gbr in enumerate(gbrs):
    gbr.fit(X, y)
    residuals[gbr.n_estimators] = y -
gbr.predict(X)
```

The following is the output:



It's a bit muddled, but hopefully, it's clear that as the number of estimators increases, the error goes down. Sadly, this isn't a panacea; first, we don't test against a holdout set, and second, as the number of estimators goes up, the training time takes longer. This isn't a big deal on the dataset we use here, but imagine one or two magnitudes higher.

How it works...

The first parameter, and the one we already looked at, is `n_estimators`—the number of weak learners that are used in GBR. In general, if you can get away with more (that is, have enough computational power), it is probably better. There are more nuances to the other parameters.

You should tune the `max_depth` parameter before all others. Since the individual learners are trees, `max_depth` controls how many nodes are produced for the trees. There's a subtle line between using the appropriate number of nodes that can fit the data well and using too many, which might cause overfitting.

The `loss` parameter controls the `loss` function, which determines the error. The `ls` parameter is the default, and stands for least squares. Least absolute deviation, Huber loss, and quantiles are also available.

Chapter 3. Building Models with Distance Metrics

This chapter will cover the following topics:

- Using KMeans to cluster data
- Optimizing the number of centroids
- Assessing cluster correctness
- Using MiniBatch KMeans to handle more data
- Quantizing an image with KMeans clustering
- Finding the closest objects in the feature space
- Probabilistic clustering with Gaussian Mixture Models
- Using KMeans for outlier detection
- Using k-NN for regression

Introduction

In this chapter, we'll cover clustering. Clustering is often grouped together with unsupervised techniques. These techniques assume that we do not know the outcome variable. This leads to ambiguity in outcomes and objectives in practice, but nevertheless, clustering can be useful. As we'll see, we can use clustering to "localize" our estimates in a supervised setting. This is perhaps why clustering is so effective; it can handle a wide range of situations, and often, the results are for the lack of a better term, "sane".

We'll walk through a wide variety of applications in this chapter; from image processing to regression and outlier detection. Through these applications, we'll see that clustering can often be viewed through a probabilistic or optimization lens. Different interpretations lead to various trade-offs. We'll walk through how to fit the models here so that you have the tools to try out many models when faced with a clustering problem.

Using KMeans to cluster data

Clustering is a very useful technique. Often, we need to divide and conquer when taking actions. Consider a list of potential customers for a business. A business might need to group customers into cohorts, and then departmentalize responsibilities for these cohorts. Clustering can help facilitate the clustering process.

KMeans is probably one of the most well-known clustering algorithms and, in a larger sense, one of the most well-known unsupervised learning techniques.

Getting ready

First, let's walk through some simple clustering, then we'll talk about how KMeans works:

```
>>> from sklearn.datasets import make_blobs  
>>> blobs, classes = make_blobs(500, centers=3)
```

Also, since we'll be doing some plotting, import matplotlib as shown:

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

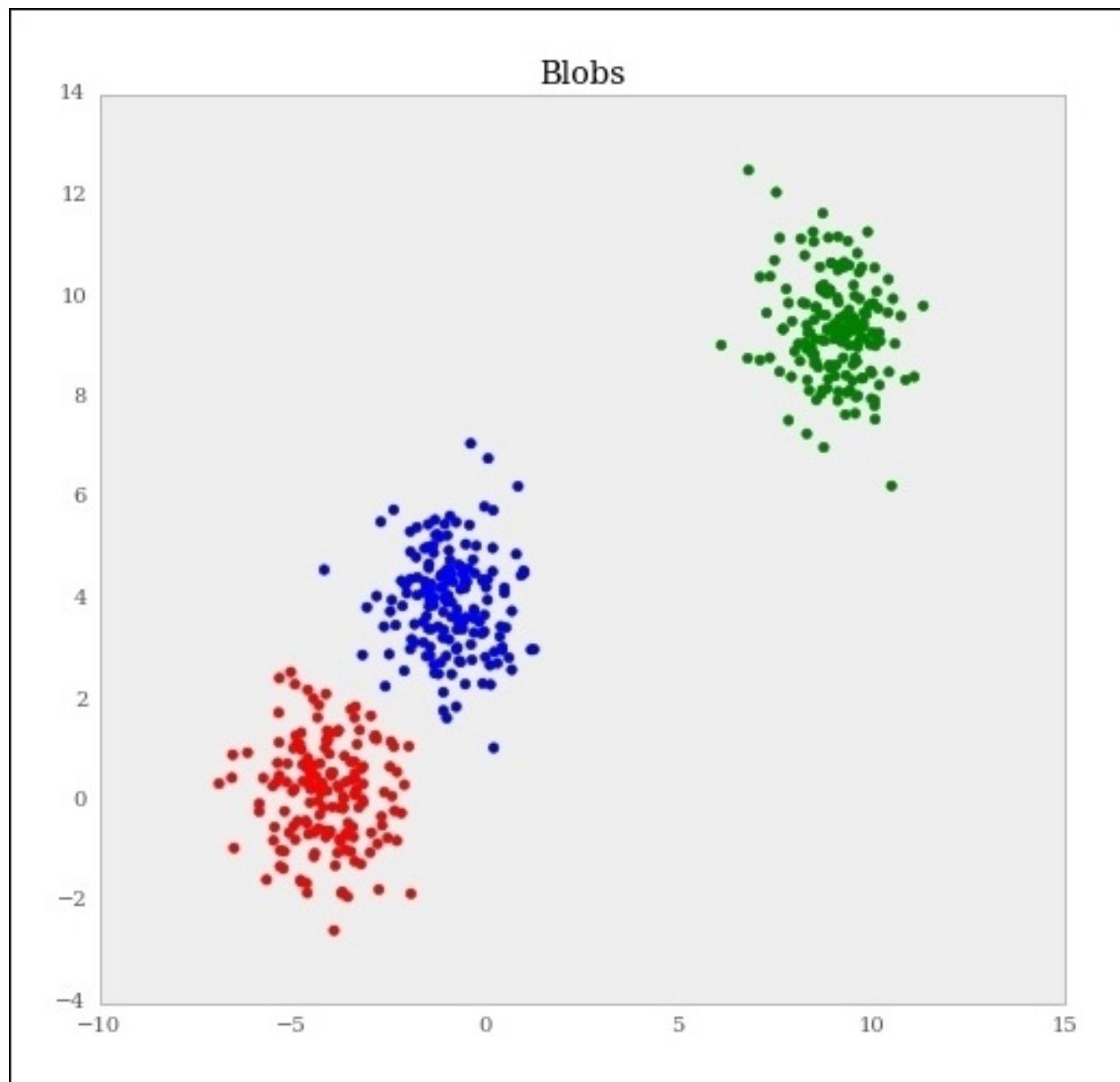
How to do it...

We are going to walk through a simple example that clusters blobs of fake data. Then we'll talk a little bit about how KMeans works to find the optimal number of blobs.

Looking at our blobs, we can see that there are three distinct clusters:

```
>>> f, ax = plt.subplots(figsize=(7.5, 7.5))
>>> ax.scatter(blobs[:, 0], blobs[:, 1],
color=rgb[classes])
>>> rgb = np.array(['r', 'g', 'b'])
>>> ax.set_title("Blobs")
```

The output is as follows:



Now we can use KMeans to find the centers of these clusters. In the first example, we'll pretend we know that there are three centers:

```
>>> from sklearn.cluster import KMeans
```

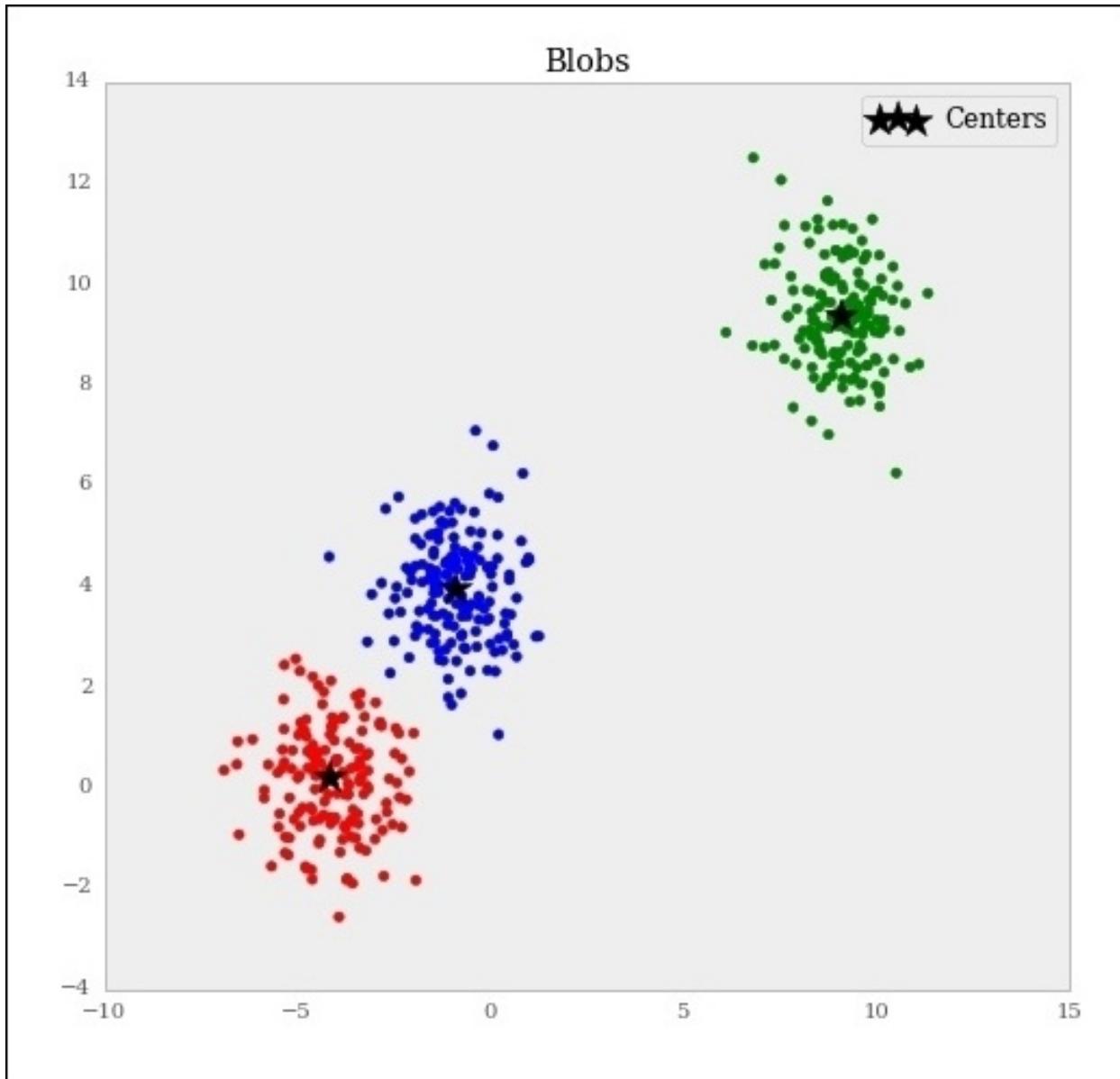
```
>>> kmean = KMeans(n_clusters=3)
>>> kmean.fit(blobs)
KMeans(copy_x=True, init='k-means++',
max_iter=300, n_clusters=3,
n_init=10, n_jobs=1,
precompute_distances=True,
random_state=None, tol=0.0001, verbose=0)

>>> kmean.cluster_centers_
array([[ 0.47819567,  1.80819197],
[ 0.08627847,  8.24102715],
[ 5.2026125 ,  7.86881767]])

>>> f, ax = plt.subplots(figsize=(7.5, 7.5))
>>> ax.scatter(blobs[:, 0], blobs[:, 1],
color=rgb[classes])
>>> ax.scatter(kmean.cluster_centers_[:, 0],
kmean.cluster_centers_[:, 1],
marker='*', s=250,
color='black', label='Centers')

>>> ax.set_title("Blobs")
>>> ax.legend(loc='best')
```

The following screenshot shows the output:



Other attributes are useful too. For instance, the `labels_` attribute will produce the expected label for each point:

```
>>> kmean.labels_[:5]
array([1, 1, 2, 2, 1], dtype=int32)
```

We can check whether `kmean.labels_` is the same as `classes`, but because KMeans has no knowledge of the classes going in, it cannot assign the sample index values to both classes:

```
>>> classes[:5]
array([0, 0, 2, 2, 0])
```

Feel free to swap `1` and `0` in `classes` to see if it matches up with `labels_`.

The `transform` function is quite useful in the sense that it will output the distance between each point and centroid:

```
>>> kmean.transform(blobs)[:5]
array([[ 6.47297373,  1.39043536,  6.4936008 ],
       [ 6.78947843,  1.51914705,  3.67659072],
       [ 7.24414567,  5.42840092,  0.76940367],
       [ 8.56306214,  5.78156881,  0.89062961],
       [ 7.32149254,  0.89737788,  5.12246797]])
```

How it works...

KMeans is actually a very simple algorithm that works to minimize the within-cluster sum of square distances from the mean. We'll be minimizing the sum of squares yet again!

It does this by first setting a pre-specified number of clusters, K , and then alternating between the following:

- Assigning each observation to the nearest cluster
- Updating each centroid by calculating the mean of each observation assigned to this cluster

This happens until some specified criterion is met.

Optimizing the number of centroids

Centroids are difficult to interpret, and it can also be very difficult to determine whether we have the correct number of centroids. It's important to understand whether your data is unlabeled or not as this will directly influence the evaluation measures we can use.

Getting ready

Evaluating the model performance for unsupervised techniques is a challenge. Consequently, `sklearn` has several methods to evaluate clustering when a ground truth is known, and very few for when it isn't.

We'll start with a single cluster model and evaluate its similarity. This is more for the purpose of mechanics as measuring the similarity of one cluster count is clearly not useful in finding the ground truth number of clusters.

How to do it...

To get started we'll create several blobs that can be used to simulate clusters of data:

```
>>> from sklearn.datasets import make_blobs
>>> import numpy as np
>>> blobs, classes = make_blobs(500, centers=3)

>>> from sklearn.cluster import KMeans
>>> kmean = KMeans(n_clusters=3)
>>> kmean.fit(blobs)
KMeans(copy_x=True, init='k-means++',
max_iter=300, n_clusters=3,
    n_init=10, n_jobs=1,
precompute_distances=True,
    random_state=None, tol=0.0001, verbose=0)
```

First, we'll look at silhouette distance. **Silhouette distance** is the ratio of the difference between in-cluster dissimilarity, the closest out-of-cluster dissimilarity, and the maximum of these two values. It can be thought of as a measure of how separate the clusters are.

Let's look at the distribution of distances from the points to the cluster centers; it's useful to understand silhouette distances:

```
>>> from sklearn import metrics
>>> silhouette_samples =
```

```

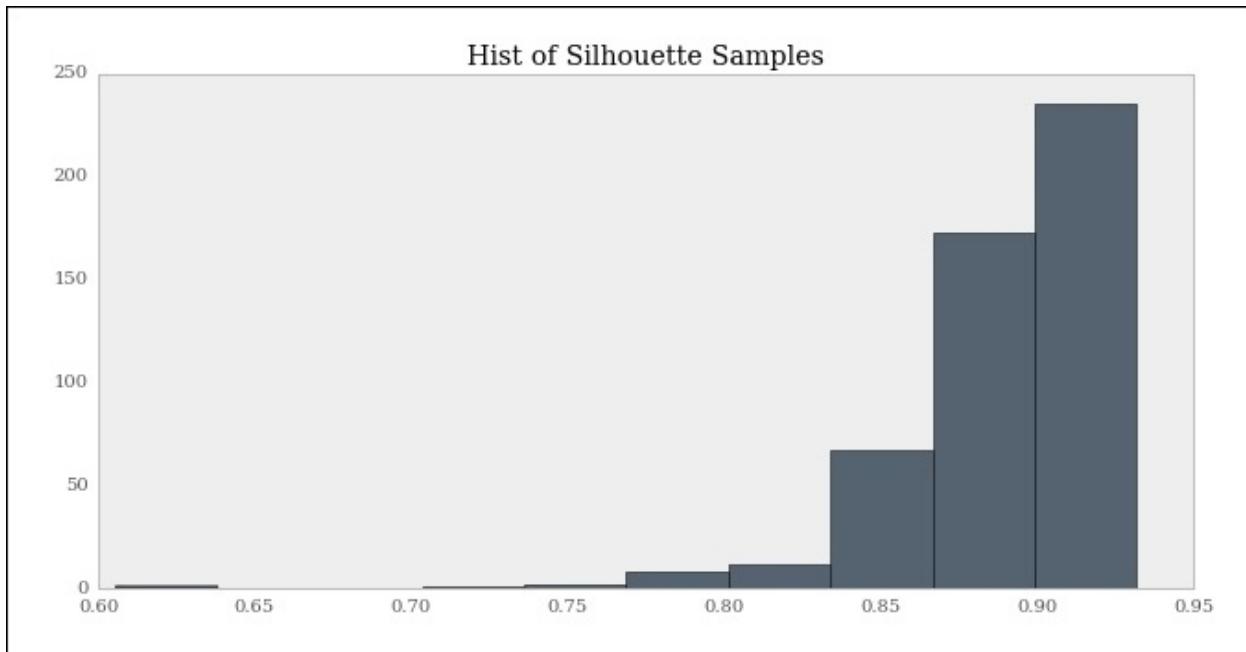
metrics.silhouette_samples(blobs,
                           kmean.labels_)
>>> np.column_stack((classes[:5],
silhouette_samples[:5]))

array([[ 1.,  0.87617292],
       [ 1.,  0.89082363],
       [ 1.,  0.88544994],
       [ 1.,  0.91478369],
       [ 1.,  0.91308287]])
>>> f, ax = plt.subplots(figsize=(10, 5))

>>> ax.set_title("Hist of Silhouette Samples")
>>> ax.hist(silhouette_samples)

```

The following is the output:



Notice that generally the higher the number of coefficients are closer to 1 (which is good) the better the score.

How it works...

The average of the silhouette coefficients is often used to describe the entire model's fit:

```
>>> silhouette_samples.mean()  
0.57130462953339578
```

It's very common; in fact, the metrics module exposes a function to arrive at the value we just got:

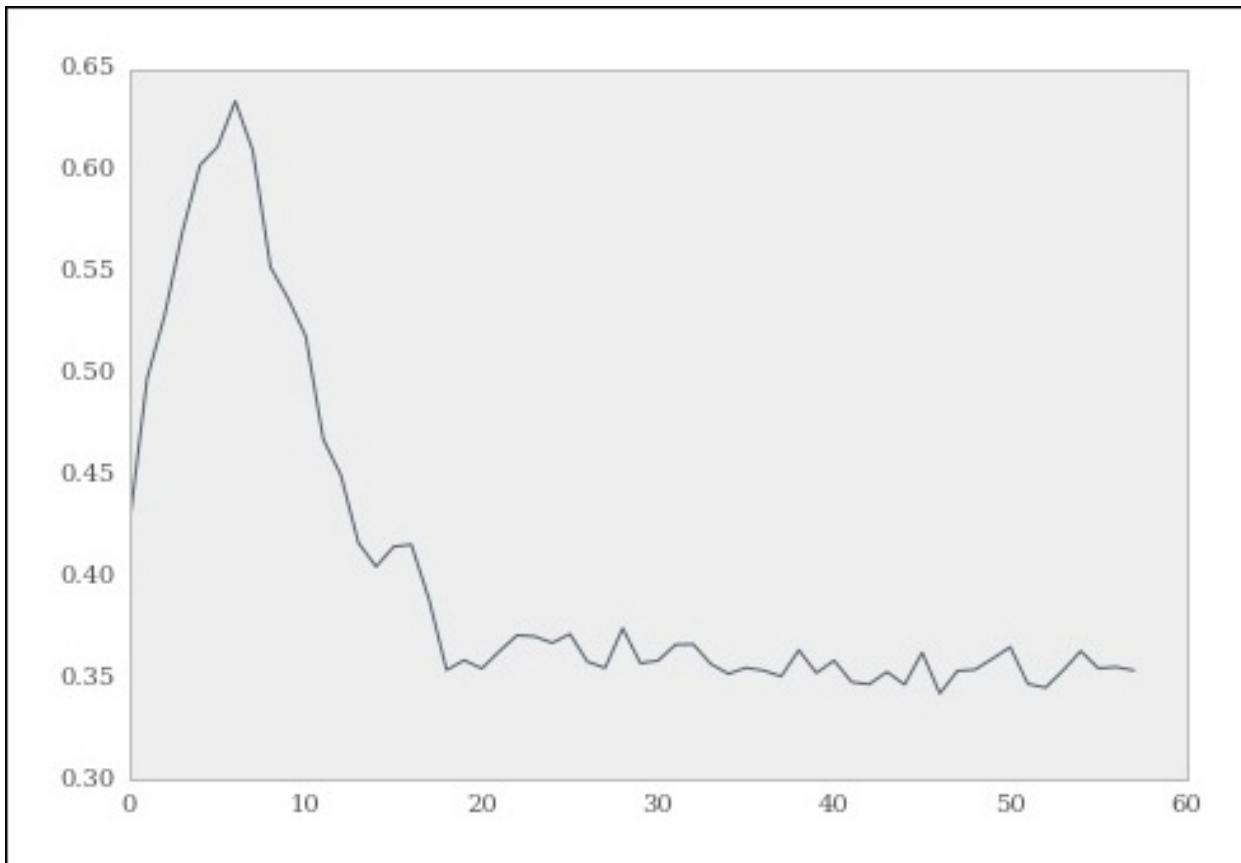
```
>>> metrics.silhouette_score(blobs,  
kmean.labels_)  
0.57130462953339578
```

Now, let's fit the models of several cluster counts and see what the average silhouette score looks like:

```
# first new ground truth  
>>> blobs, classes = make_blobs(500, centers=10)  
>>> sillhouette_avgs = []  
  
# this could take a while  
>>> for k in range(2, 60):  
    kmean = KMeans(n_clusters=k).fit(blobs)  
  
    sillhouette_avgs.append(metrics.silhouette_score  
(blobs,  
                                         kmean.labels_))  
  
>>> f, ax = plt.subplots(figsize=(7, 5))
```

```
>>> ax.plot(sillhouette_avgs)
```

The following is the output:



This plot shows that the silhouette averages as the number of centroids increase. We can see that the optimum number, according to the data generating process, is 3, but here it looks like it's around 6 or 7. This is the reality of clustering; quite often, we won't get the correct number of clusters, we can only really hope to estimate the number of clusters to some approximation.

Assessing cluster correctness

We talked a little bit about assessing clusters when the ground truth is not known. However, we have not yet talked about assessing KMeans when the cluster is known. In a lot of cases, this isn't knowable; however, if there is outside annotation, we will know the ground truth, or at least the proxy, sometimes.

Getting ready

So, let's assume a world where we have some outside agent supplying us with the ground truth.

We'll create a simple dataset, evaluate the measures of correctness against the ground truth in several ways, and then discuss them:

```
>>> from sklearn import datasets
>>> from sklearn import cluster
>>> blobs, ground_truth =
datasets.make_blobs(1000, centers=3,
cluster_std=1.75)
```

How to do it...

Before we walk through the metrics, let's take a look at the dataset:

```
>>> f, ax = plt.subplots(figsize=(7, 5))

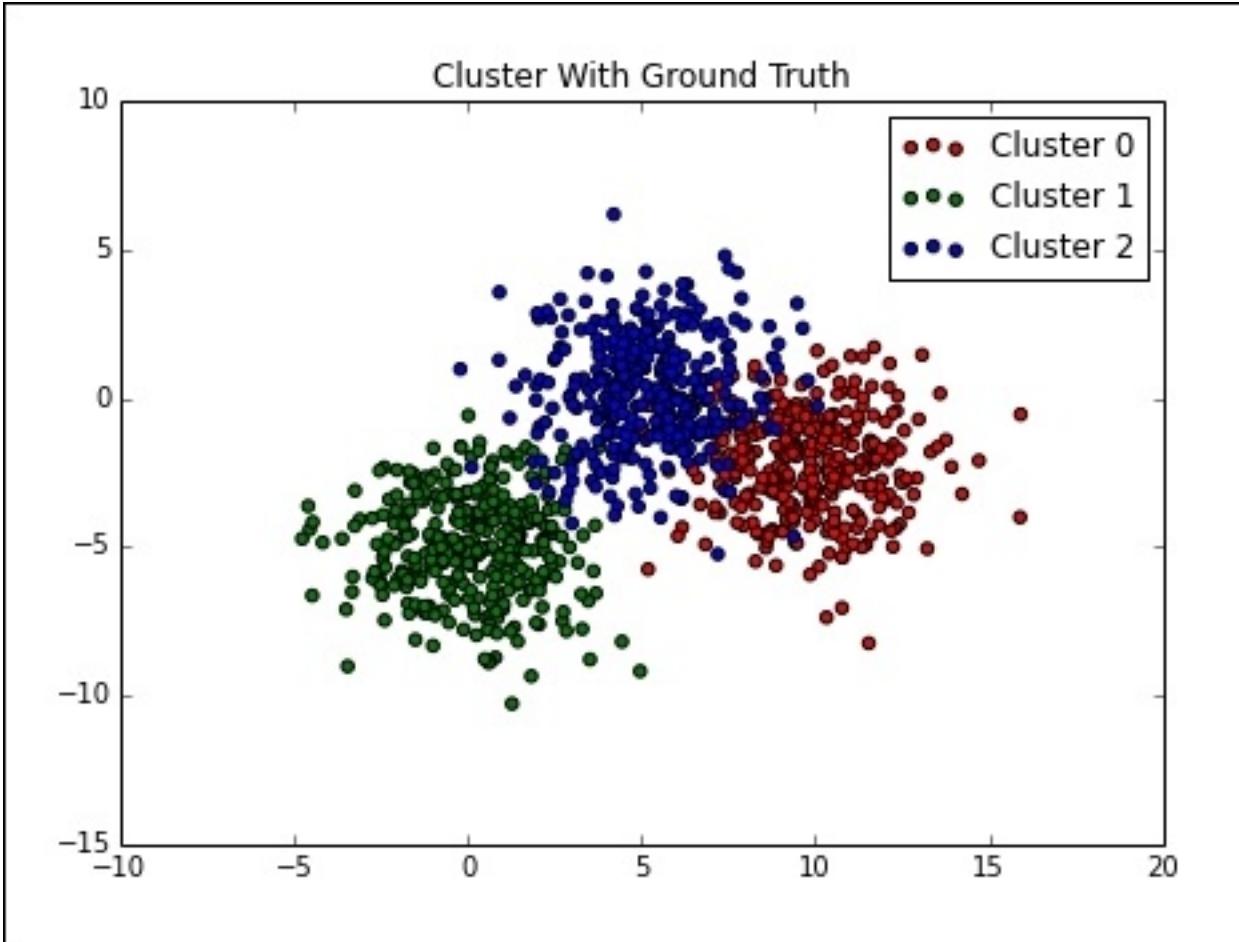
>>> colors = ['r', 'g', 'b']

>>> for i in range(3):
    p = blobs[ground_truth == i]
    ax.scatter(p[:,0], p[:,1], c=colors[i],
               label="Cluster {}".format(i))

>>> ax.set_title("Cluster With Ground Truth")
>>> ax.legend()

>>> f.savefig("9485OS_03-16")
```

The following is the output:



In order to fit a KMeans model we'll create a KMeans object from the `cluster` module:

```
>>> kmeans = cluster.KMeans(n_clusters=3)

>>> kmeans.fit(blobs)

KMeans(copy_x=True, init='k-means++',
max_iter=300, n_clusters=3,
n_init=10, n_jobs=1,
```

```
precompute_distances=True,
    random_state=None, tol=0.0001, verbose=0)

>>> kmeans.cluster_centers_

array([[ 5.18993766,  0.35110059],
       [ 0.18300097, -4.9480336 ],
       [ 10.01421381, -2.26274328]])
```

Now that we've fit the model, let's have a look at the cluster centroids:

```
>>> f, ax = plt.subplots(figsize=(7, 5))

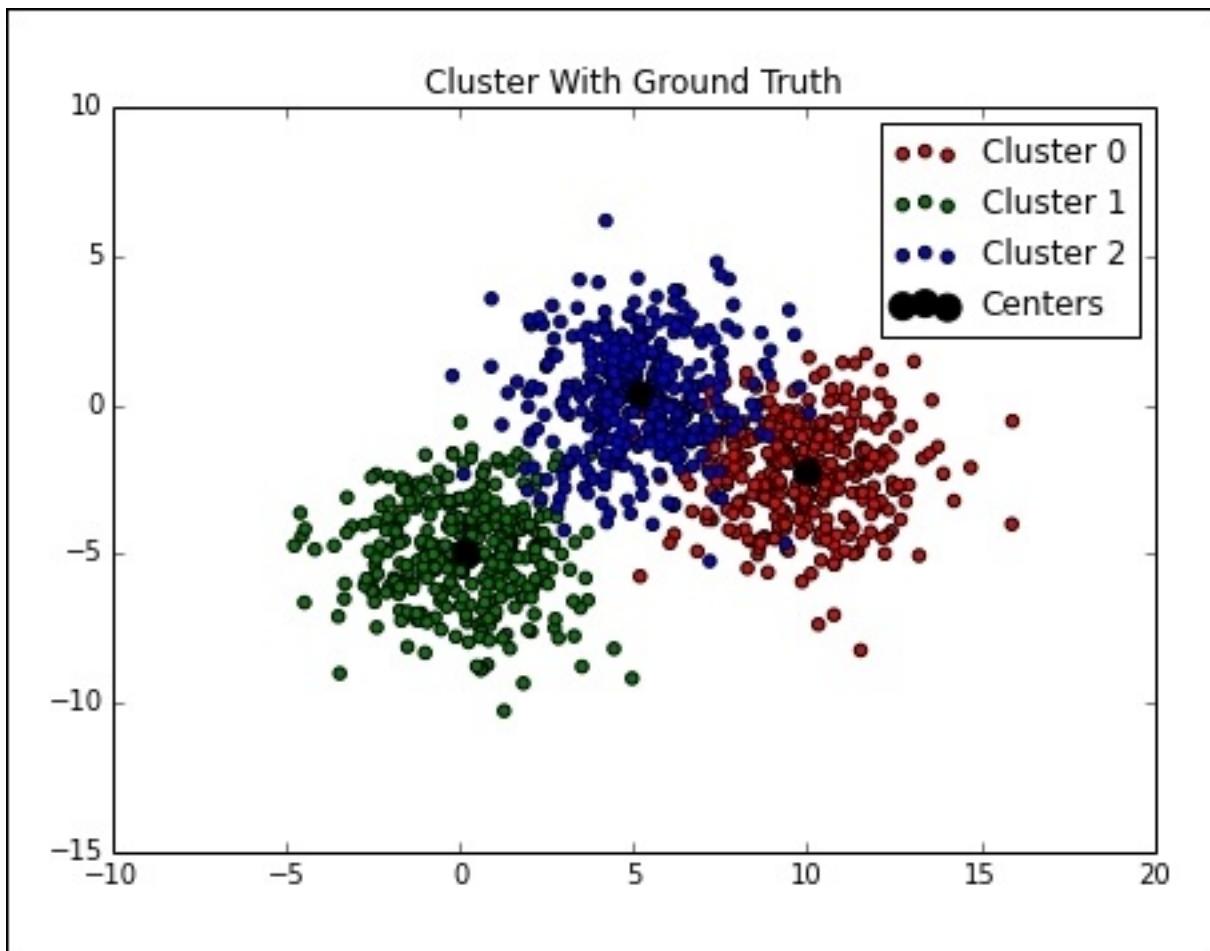
>>> colors = ['r', 'g', 'b']

>>> for i in range(3):
    p = blobs[ground_truth == i]
    ax.scatter(p[:,0], p[:,1], c=colors[i],
               label="Cluster {}".format(i))

>>> ax.scatter(kmeans.cluster_centers_[:, 0],
              kmeans.cluster_centers_[:, 1],
              s=100, color='black',
              label='Centers')
>>> ax.set_title("Cluster With Ground Truth")
>>> ax.legend()

>>> f.savefig("9485OS_03-17")
```

The following is the output:



Now that we can view the clustering performance as a classification exercise, the metrics that are useful in its context are also useful here:

```
>>> for i in range(3):
    print (kmeans.labels_ == ground_truth)
[ground_truth == i]
    .astype(int).mean()
```

0.0778443113772

```
0.990990990991  
0.0570570570571
```

Clearly, we have some backward clusters. So, let's get this straightened out first, and then we'll look at the accuracy:

```
>>> new_ground_truth = ground_truth.copy()  
  
>>> new_ground_truth[ground_truth == 0] = 2  
>>> new_ground_truth[ground_truth == 2] = 0  
  
>>> for i in range(3):  
    print (kmeans.labels_ ==  
new_ground_truth)[ground_truth == i]  
  
.astype(int).mean()
```

```
0.919161676647  
0.990990990991  
0.90990990991
```

So, we're roughly correct 90 percent of the time. The second measure of similarity we'll look at is the mutual information score:

```
>>> from sklearn import metrics  
  
>>>  
metrics.normalized_mutual_info_score(ground_trut  
h, kmeans.labels_)  
  
0.78533737204433651
```

As the score tends to be 0, the label assignments are probably not generated through similar processes; however, the score being closer to 1 means that there is a large amount of agreement between the two labels.

For example, let's look at what happens when the mutual information score itself:

```
>>>  
metrics.normalized_mutual_info_score(ground_trut  
h, ground_truth)  
  
1.0
```

Given the name, we can tell that there is probably an unnormalized `mutual_info_score`:

```
>>> metrics.mutual_info_score(ground_truth,  
kmeans.labels_ )  
  
0.78945287371677486
```

These are very close; however, normalized mutual information is the mutual information divided by the root of the product of the entropy of each set truth and assigned label.

There's more...

One cluster metric we haven't talked about yet and one that is not reliant on the ground truth is inertia. It is not very well documented as a metric at the moment. However, it is the metric that KMeans minimizes.

Inertia is the sum of the squared difference between each point and its assigned cluster. We can use a little NumPy to determine this:

```
>>> kmeans.inertia_
```

Using MiniBatch KMeans to handle more data

KMeans is a nice method to use; however, it is not ideal for a lot of data. This is due to the complexity of KMeans. This said, we can get approximate solutions with much better algorithmic complexity using KMeans.

Getting ready

MiniBatch KMeans is a faster implementation of KMeans. KMeans is computationally very expensive; the problem is **NP-hard**.

However, using MiniBatch KMeans, we can speed up KMeans by orders of magnitude. This is achieved by taking many subsamples that are called MiniBatches. Given the convergence properties of subsampling, a close approximation to regular KMeans is achieved, given good initial conditions.

How to do it...

Let's do some very high-level profiling of MiniBatch clustering. First, we'll look at the overall speed difference, and then we'll look at the errors in the estimates:

```
>>> from sklearn.datasets import make_blobs  
>>> blobs, labels = make_blobs(int(1e6), 3)  
  
>>> from sklearn.cluster import KMeans,  
MiniBatchKMeans  
  
>>> kmeans = KMeans(n_clusters=3)  
>>> minibatch = MiniBatchKMeans(n_clusters=3)
```

Tip

Understand that these metrics are meant to expose the issue. Therefore, great care is taken to ensure the highest accuracy of the benchmarks. There is a lot of information available on this topic; if you really want to get to the heart of why MiniBatch KMeans is better at scaling, it will be a good idea to review what's available.

Now that the setup is complete, we can measure the time difference:

```
>>> %time kmeans.fit(blobs) #IPython Magic  
CPU times: user 8.17 s, sys: 881 ms, total: 9.05  
s Wall time: 9.97 s
```

```
>>> %time minibatch.fit(blobs)
CPU times: user 4.04 s, sys: 90.1 ms, total:
4.13 s Wall time: 4.69 s
```

There's a large difference in CPU times. The difference in clustering performance is shown as follows:

```
>>> kmeans.cluster_centers_[0]
array([ 1.10522173, -5.59610761, -8.35565134])

>>> minibatch.cluster_centers_[0]
array([ 1.12071187, -5.61215116, -8.32015587])
```

The next question we might ask is how far apart the centers are:

```
>>> from sklearn.metrics import pairwise
>>>
pairwise.pairwise_distances(kmeans.cluster_center_
rs_[0] ,
minibatch.cluster_centers_[0])
array([[ 0.03305309]])
```

This seems to be very close. The diagonals will contain the cluster center differences:

```
>>>
np.diag(pairwise.pairwise_distances(kmeans.clust
er_centers_ ,
```

```
    minibatch.cluster_centers_))
array([ 0.04191979,  0.03133651,  0.04342707])
```

How it works...

The batches here are key. Batches are iterated through to find the batch mean; for the next iteration, the prior batch mean is updated in relation to the current iteration. There are several options that dictate the general KMeans' behavior and parameters that determine how MiniBatch KMeans gets updated.

The `batch_size` parameter determines how large the batches should be. Just for fun, let's run MiniBatch; however, this time we set the batch size the same as the dataset size:

```
>>> minibatch =  
MiniBatchKMeans(batch_size=len(blobs))  
>>> %time minibatch.fit(blobs)  
CPU times: user 34.6 s, sys: 3.17 s, total: 37.8  
s Wall time: 44.6 s
```

Clearly, this is against the spirit of the problem, but it does illustrate an important point. Choosing poor initial conditions can affect how well models, particularly clustering models, converge. With MiniBatch KMeans, there is no guarantee that the global optimum will be achieved.

Quantizing an image with KMeans clustering

Image processing is an important topic in which clustering has some application. It's worth pointing out that there are several very good image-processing libraries in Python. **scikit-image** is a "sister" project of scikit-learn. It's worth taking a look at if you want to do anything complicated.

Getting ready

We will have some fun in this recipe. The goal is to use cluster to blur an image.

First, we'll make use of SciPy to read the image. The image is translated in a 3-dimensional array; the x and y coordinates describe the height and width, and the third dimension represents the RGB values for each image:

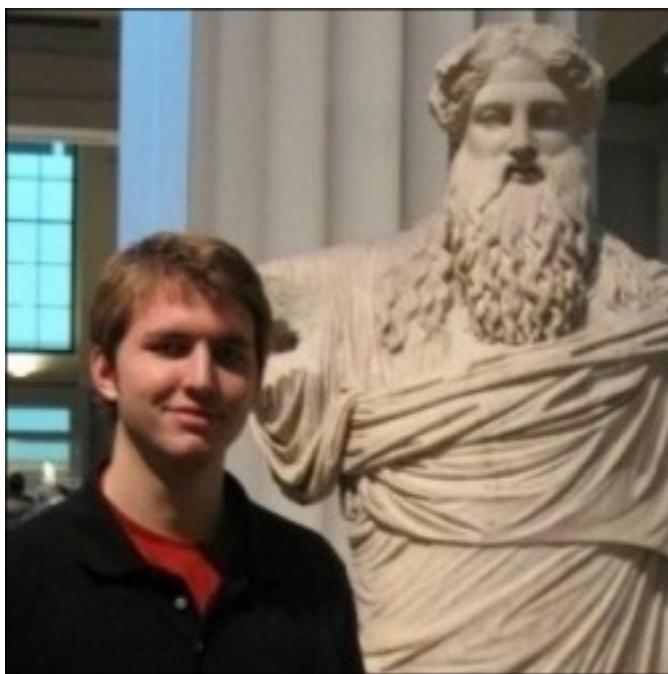
```
# in your terminal
$ wget
http://blog.trenthauck.com/assets/headshot.jpg
```

How do it...

Now, let's read the image in Python:

```
>>> from scipy import ndimage  
>>> img = ndimage.imread("headshot.jpg")  
>>> plt.imshow(img)
```

The following image is seen:



Hey, that's (a younger) me!

Now that we have the image, let's check its dimensions:

```
>>> img.shape
```

```
(420, 420, 3)
```

To actually quantize the image, we need to convert it into a two-dimensional array, with the length being 420 x 420 and the width being the RGB values. A better way to think about this is to have a bunch of data points in three-dimensional space and cluster the points to reduce the number of distant colors in the image—a simple way to put quantization.

First, let's reshape our array; it is a NumPy array, and thus trivial to work with:

```
>>> x, y, z = img.shape
>>> long_img = img.reshape(x*y, z)
>>> long_img.shape
(176400, 3)
```

Now we can start the clustering process. First, let's import the cluster module and create a KMeans object. We'll pass `n_clusters=5` so that we have five clusters, or really, five distinct colors.

This will be a good recipe to practice using silhouette distance that we reviewed in the *Optimizing the number of centroids* recipe:

```
>>> from sklearn import cluster
>>> k_means = cluster.KMeans(n_clusters=5)
```

```
>>> k_means.fit(long_img)
```

Now that we have our fit KMeans objects, let's take a look at our colors:

```
>>> centers = k_means.cluster_centers_
>>> centers
array([[ 142.58775848, 206.12712986,
226.04416873],
       [ 86.29356543, 68.86312505,
54.04770507],
       [ 194.36182899, 172.19845258,
149.65603813],
       [ 24.67768412, 20.45778933,
16.19698314],
       [ 149.27801776, 132.19850659,
115.32729167]])
```

How it works...

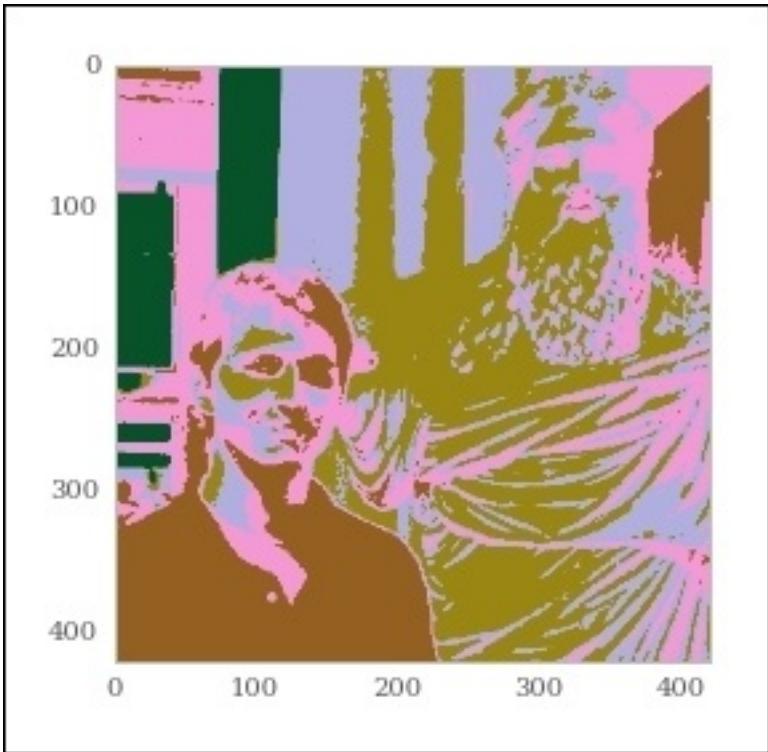
Now that we have the centers, the next thing we need is the labels. This will tell us which points should be associated with which clusters:

```
>>> labels = k_means.labels_
>>> labels[:5]
array([1, 1, 1, 1, 1], dtype=int32)
```

At this point, we require the simplest of NumPy array manipulation followed by a bit of reshaping, and we'll have the new image:

```
>>> plt.imshow(centers[labels].reshape(x, y, z))
```

The following is the resultant image:



Finding the closest objects in the feature space

Sometimes, the easiest thing to do is to just find the distance between two objects. We just need to find some distance metric, compute the pairwise distances, and compare the outcomes to what's expected.

Getting ready

A lower-level utility in scikit-learn is `sklearn.metrics.pairwise`. This contains server functions to compute the distances between the vectors in a matrix X or the distances between the vectors in X and Y easily.

This can be useful for information retrieval. For example, given a set of customers with attributes of X , we might want to take a reference customer and find the closest customers to this customer. In fact, we might want to rank customers by the notion of similarity measured by a distance function. The quality of the similarity depends upon the feature space selection as well as any transformation we might do on the space.

We'll walk through several different scenarios of measuring distance.

How to do it...

We will use the `pairwise_distances` function to determine the "closeness" of objects. Remember that the closeness is really just similarity that we use our distance function to grade.

First, let's import the pairwise distance function from the `metrics` module and create a dataset to play with:

```
>>> from sklearn.metrics import pairwise
>>> from sklearn.datasets import make_blobs
>>> points, labels = make_blobs()
```

This simplest way to check the distances is `pairwise_distances`:

```
>>> distances =
pairwise.pairwise_distances(points)
```

`distances` is an $N \times N$ matrix with 0s along the diagonals. In the simplest case, let's see the distances between each point and the first point:

```
>>> np.diag(distances) [:5]
array([ 0.,  0.,  0.,  0.,  0.])
```

Now we can look for points that are closest to the first point in `points`:

```
>>> distances[0][:5]
array([ 0., 11.82643041, 1.23751545, 1.17612135,
       14.61927874])
```

Ranking the points by closeness is very easy with `np.argsort`:

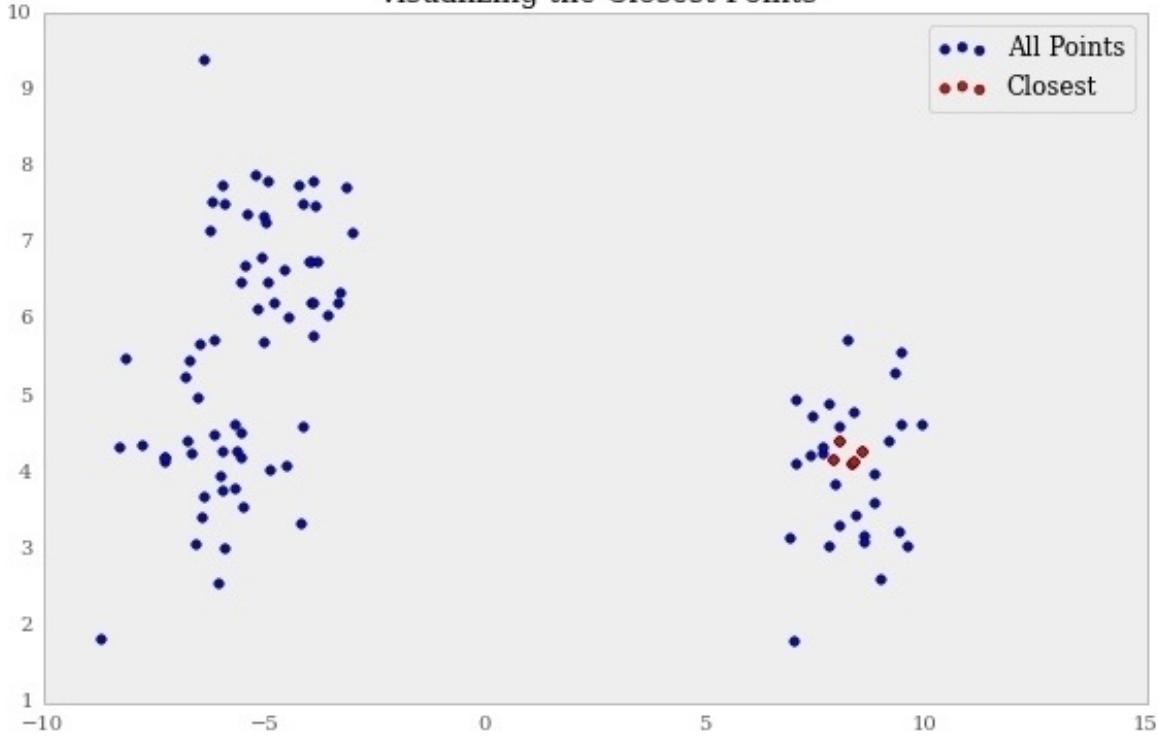
```
>>> ranks = np.argsort(distances[0])
>>> ranks[:5]
array([ 0, 27, 98, 23, 67])
```

The great thing about `argsort` is that now we can sort our points matrix to get the actual points:

```
>>> points[ranks][:5]
array([[ 8.96147382, -1.90405304],
       [ 8.75417014, -1.76289919],
       [ 8.78902665, -2.27859923],
       [ 8.59694131, -2.10057667],
       [ 8.70949958, -2.30040991]])
```

It's useful to see what the closest points look like. Other than some assurances, this works as intended:

Visualizing the Closest Points



How it works...

Given some distance function, each point is measured in a pairwise function. The default is the Euclidian distance, which is as follows:

$$d(x, y) = \sqrt{\sum_i (x_i - y_i)^2}$$

Verbally, this takes the difference between each component of the two vectors, squares the difference, sums them, and then takes the square root. This looks very familiar as we used something very similar to this when looking at the mean-squared error. If we take the square root, we have the same thing. In fact, a metric used often is **root-mean-square deviation (RMSE)**, which is just the applied distance function.

In Python, this looks like the following:

```
>>> def euclid_distances(x, y):
    return np.power(np.power(x - y, 2).sum(), .5)
>>> euclid_distances(points[0], points[1])
11.826430406213145
```

There are several other functions available in scikit-learn, but scikit-learn will also use distance functions of SciPy. At the time of writing this book, the scikit-learn distance functions support sparse matrixes. Check out the SciPy documentation for more information on the distance functions:

- cityblock
- cosine
- euclidean
- l1
- l2
- manhattan

We can now solve problems. For example, if we were standing on a grid at the origin, and the lines were the streets, how far will we have to travel to get to point (5, 5)?.

```
>>> pairwise.pairwise_distances([[0, 0], [5, 5]], metric='cityblock')[0]
array([ 0., 10.])
```

There's more...

Using pairwise distances, we can find the similarity between bit vectors. It's a matter of finding the hamming distance, which is defined as follows:

$$\sum_i I_{x_i \neq y_i}$$

Use the following command:

```
>>> x = np.random.binomial(1, .5, size=(2, 4)).astype(np.bool)
>>> x
array([[False,  True, False, False],
       [False, False, False,  True]], dtype=bool)

>>> pairwise.pairwise_distances(x, metric='hamming')
array([[ 0. ,  0.25],
       [ 0.25,  0. ]])
```

Probabilistic clustering with Gaussian Mixture Models

In KMeans, we assume that the variance of the clusters is equal. This leads to a subdivision of space that determines how the clusters are assigned; but, what about a situation where the variances are not equal and each cluster point has some probabilistic association with it?

Getting ready

There's a more probabilistic way of looking at KMeans clustering. Hard KMeans clustering is the same as applying a Gaussian Mixture Model with a covariance matrix, S , which can be factored to the error times of the identity matrix. This is the same covariance structure for each cluster. It leads to **spherical clusters**.

However, if we allow S to vary, a GMM can be estimated and used for prediction. We'll look at how this works in a univariate sense, and then expand to more dimensions.

How to do it...

First, we need to create some data. For example, let's simulate heights of both women and men. We'll use this example throughout this recipe. It's a simple example, but hopefully, will illustrate what we're trying to accomplish in an N dimensional space, which is a little easier to visualize:

```
>>> import numpy as np
>>> N = 1000

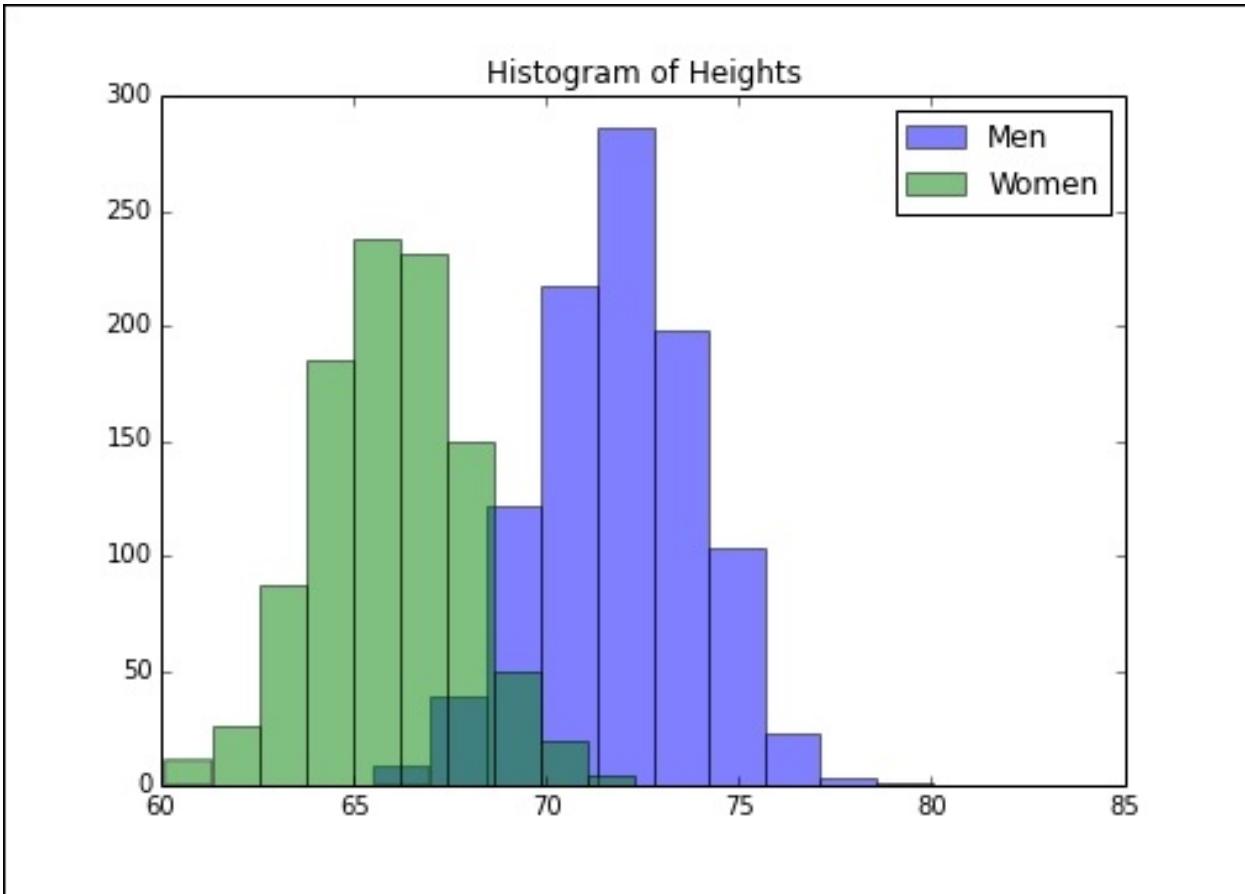
>>> in_m = 72
>>> in_w = 66

>>> s_m = 2
>>> s_w = s_m

>>> m = np.random.normal(in_m, s_m, N)
>>> w = np.random.normal(in_w, s_w, N)
>>> from matplotlib import pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.set_title("Histogram of Heights")
>>> ax.hist(m, alpha=.5, label="Men");
>>> ax.hist(w, alpha=.5, label="Women");
>>> ax.legend()
```

The following is the output:



Next, we might be interested in subsampling the group, fitting the distribution, and then predicting the remaining groups:

```
>>> random_sample = np.random.choice([True, False], size=m.size)
>>> m_test = m[random_sample]
>>> m_train = m[~random_sample]

>>> w_test = w[random_sample]
>>> w_train = w[~random_sample]
```

Now we need to get the empirical distribution of the heights of both men and women based on the training set:

```
>>> from scipy import stats  
>>> m_pdf = stats.norm(m_train.mean(),  
m_train.std())  
>>> w_pdf = stats.norm(w_train.mean(),  
w_train.std())
```

For the test set, we will calculate based on the likelihood that the data point was generated from either distribution, and the most likely distribution will get the appropriate label assigned. We will, of course, look at how accurate we were:

```
>>> m_pdf.pdf(m[0])  
0.043532673457165431  
  
>>> w_pdf.pdf(m[0])  
9.2341848872766183e-07
```

Notice the difference in likelihoods.

Assume that we guess situations when the men's probability is higher, but we overwrite them if the women's probability is higher:

```
>>> guesses_m = np.ones_like(m_test)  
>>> guesses_m[m_pdf.pdf(m_test) <  
w_pdf.pdf(m_test)] = 0
```

Obviously, the question is how accurate we are. Since `guesses_m` will be 1 if we are correct, and 0 if we aren't, we take the mean of the vector and get the accuracy:

```
>>> guesses_m.mean()
0.93775100401606426
```

Not too bad! Now, to see how well we did with for the women's group, use the following commands:

```
>>> guesses_w = np.ones_like(w_test)
>>> guesses_w[m_pdf.pdf(w_test) >
w_pdf.pdf(w_test)] = 0
>>> guesses_w.mean()
0.93172690763052213
```

Let's allow the variance to differ between groups. First, create some new data:

```
>>> s_m = 1
>>> s_w = 4

>>> m = np.random.normal(in_m, s_m, N)
>>> w = np.random.normal(in_w, s_w, N)
```

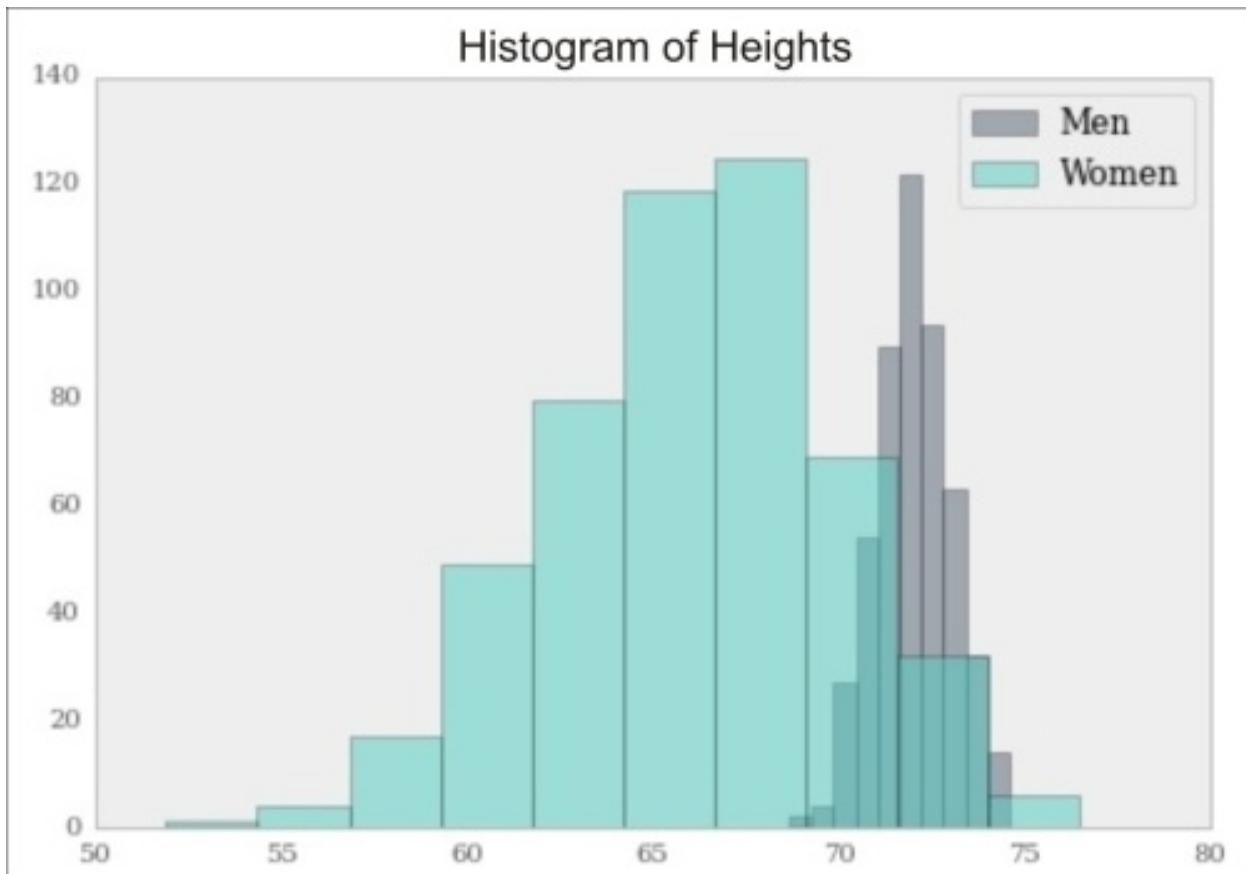
Then, create a training set:

```
>>> m_test = m[random_sample]
>>> m_train = m[~random_sample]

>>> w_test = w[random_sample]
>>> w_train = w[~random_sample]
```

```
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.set_title("Histogram of Heights")
>>> ax.hist(m_train, alpha=.5, label="Men");
>>> ax.hist(w_train, alpha=.5, label="Women");
>>> ax.legend()
```

Let's take a look at the difference in variances between the men and women:

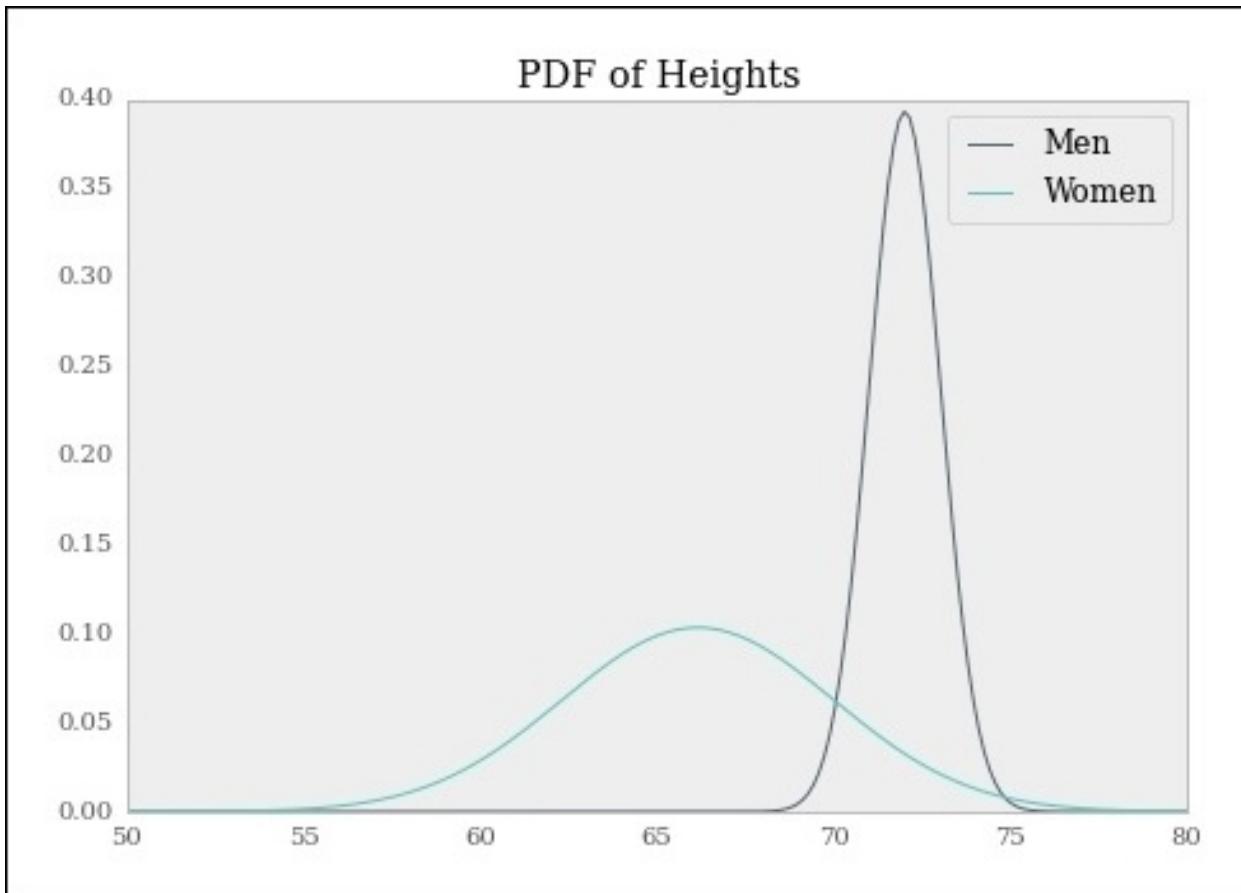


Now we can create the same PDFs:

```
>>> m_pdf = stats.norm(m_train.mean(),
```

```
m_train.std())
>>> w_pdf = stats.norm(w_train.mean(),
w_train.std())
```

The following is the output:



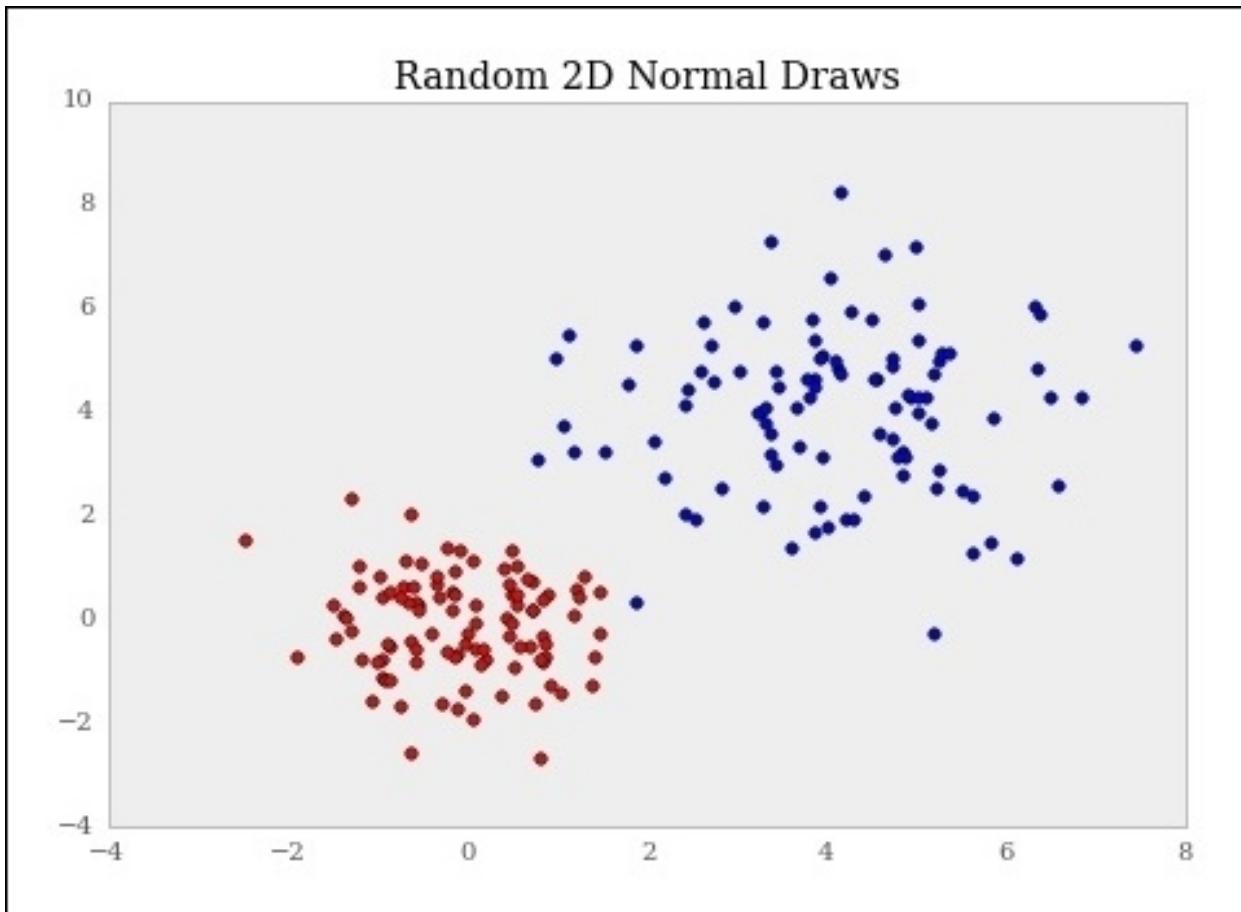
You can imagine this in a multidimensional space:

```
>>> class_A = np.random.normal(0, 1, size=(100,
2))
>>> class_B = np.random.normal(4, 1.5, size=
(100, 2))
```

```
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.scatter(class_A[:,0], class_A[:,1],
label='A', c='r')
>>> ax.scatter(class_B[:,0], class_B[:,1],
label='B')
```

The following is the output:



How it works...

Okay, so now that we've looked at how we can classify points based on distribution, let's look at how we can do this in scikit-learn:

```
>>> from sklearn.mixture import GMM  
>>> gmm = GMM(n_components=2)  
>>> X = np.row_stack((class_A, class_B))  
>>> y = np.hstack((np.ones(100), np.zeros(100)))
```

Since we're good little data scientists, we'll create a training set:

```
>>> train = np.random.choice([True, False], 200)  
>>> gmm.fit(X[train])  
GMM(covariance_type='diag', init_params='wmc',  
min_covar=0.001,  
n_components=2, n_init=1, n_iter=100,  
params='wmc',  
random_state=None, thresh=0.01)
```

Fitting and predicting is done in the same way as fitting is done for many of the other objects in scikit-learn:

```
>>> gmm.fit(X[train])  
>>> gmm.predict(X[train])[:5]  
array([0, 0, 0, 0, 0])
```

There are other methods worth looking at now that the

model has been fit.

For example, using `score_samples`, we can actually get the per-sample likelihood for each label.

Using KMeans for outlier detection

In this chapter, we'll look at both the debate and mechanics of KMeans for outlier detection. It can be useful to isolate some types of errors, but care should be taken when using it.

Getting ready

In this recipe, we'll use KMeans to do outlier detections on a cluster of points. It's important to note that there are many "camps" when it comes to outliers and outlier detection. On one hand, we're potentially removing points that were generated by the data-generating process by removing outliers. On the other hand, outliers can be due to a measurement error or some other outside factor.

This is the most credence we'll give to the debate; the rest of this recipe is about finding outliers; we'll work under the assumption that our choice to remove outliers is justified.

The act of outlier detection is a matter of finding the centroids of the clusters, and then identifying points that are potential outliers by their distances from the centroid.

How to do it...

First, we'll generate a single blob of 100 points, and then we'll identify the 5 points that are furthest from the centroid. These are the potential outliers:

```
>>> from sklearn.datasets import make_blobs  
>>> X, labels = make_blobs(100, centers=1)  
>>> import numpy as np
```

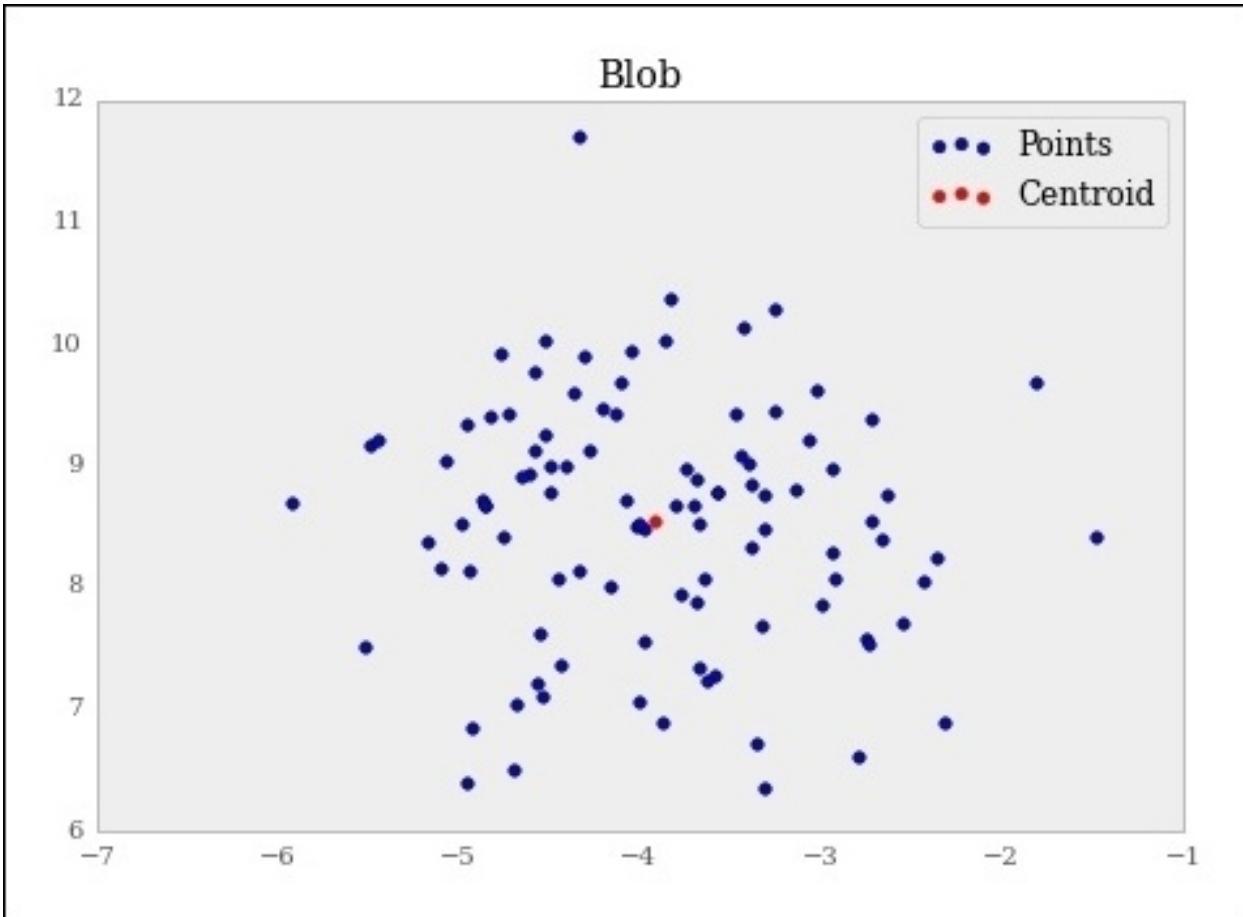
It's important that the KMeans cluster has a single center. This idea is similar to a one-class SVM that is used for outlier detection:

```
>>> from sklearn.cluster import KMeans  
>>> kmeans = KMeans(n_clusters=1)  
>>> kmeans.fit(X)
```

Now, let's look at the plot. For those playing along at home, try to guess which points will be identified as one of the five outliers:

```
>>> f, ax = plt.subplots(figsize=(7, 5))  
>>> ax.set_title("Blob")  
>>> ax.scatter(X[:, 0], X[:, 1], label='Points')  
>>> ax.scatter(kmeans.cluster_centers_[:, 0],  
                 kmeans.cluster_centers_[:, 1],  
                 label='Centroid',  
                 color='r')  
>>> ax.legend()
```

The following is the output:



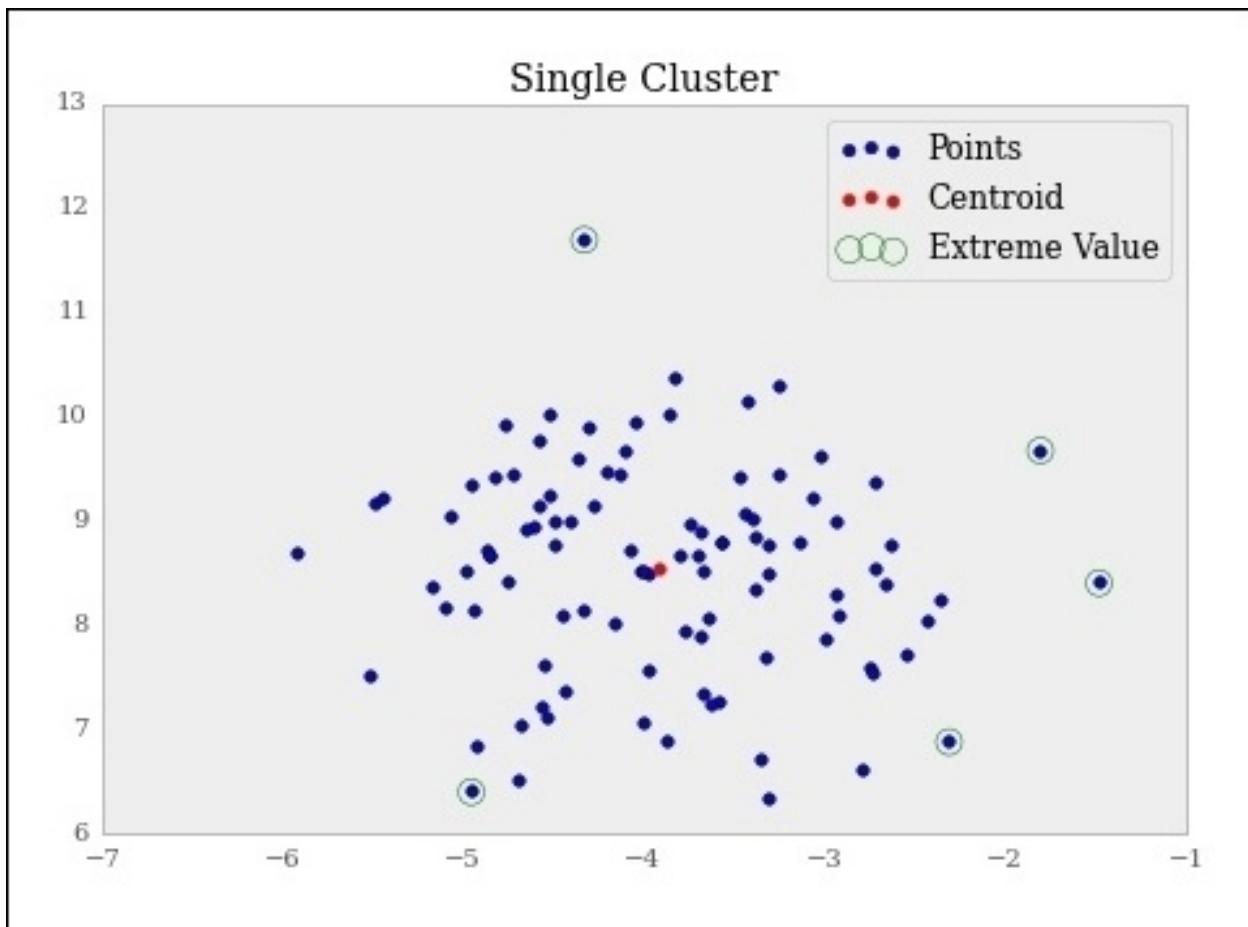
Now, let's identify the five closest points:

```
>>> distances = kmeans.transform(X)
# argsort returns an array of indexes which will
# sort the array in ascending order
# so we reverse it via [::-1] and take the top
# five with [:5]
>>> sorted_idx = np.argsort(distances.ravel())
[::-1][:5]
```

Now, let's see which plots are the farthest away:

```
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.set_title("Single Cluster")
>>> ax.scatter(X[:, 0], X[:, 1], label='Points')
>>> ax.scatter(kmeans.cluster_centers_[:, 0],
   kmeans.cluster_centers_[:, 1],
   label='Centroid', color='r')
>>> ax.scatter(X[sorted_idx][:, 0],
   X[sorted_idx][:, 1],
   label='Extreme Value',
edgecolors='g',
   facecolors='none', s=100)
>>> ax.legend(loc='best')
```

The following is the output:



It's easy to remove these points if we like:

```
>>> new_X = np.delete(X, sorted_idx, axis=0)
```

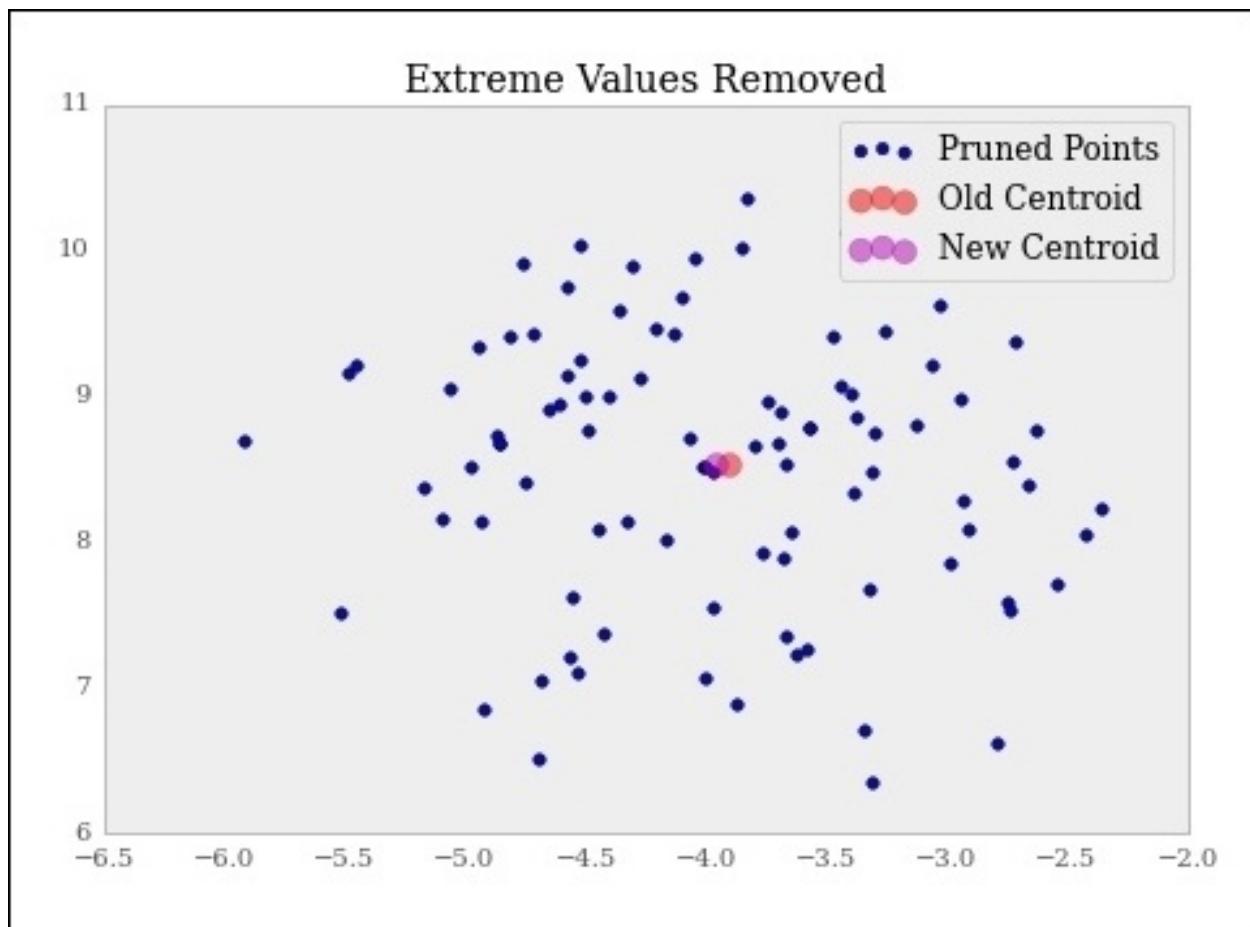
Also, the centroid clearly changes with the removal of these points:

```
>>> new_kmeans = KMeans(n_clusters=1)
>>> new_kmeans.fit(new_X)
```

Let's visualize the difference between the old and new centroids:

```
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.set_title("Extreme Values Removed")
>>> ax.scatter(new_X[:, 0], new_X[:, 1],
label='Pruned Points')
>>> ax.scatter(kmeans.cluster_centers_[:, 0],
            kmeans.cluster_centers_[:, 1],
label='Old Centroid',
            color='r', s=80, alpha=.5)
>>> ax.scatter(new_kmeans.cluster_centers_[:, 0],
            new_kmeans.cluster_centers_[:, 1], label='New Centroid',
            color='m', s=80, alpha=.5)
>>> ax.legend(loc='best')
```

The following is the output:



Clearly, the centroid hasn't moved much, which is to be expected when only removing the five most extreme values. This process can be repeated until we're satisfied that the data is representative of the process.

How it works...

As we've already seen, there is a fundamental connection between the Gaussian distribution and the KMeans clustering. Let's create an empirical Gaussian based off the centroid and sample covariance matrix and look at the probability of each point—theoretically, the five points we removed. This just shows that we have in fact removed the values with the least likelihood. This idea between distances and likelihoods is very important, and will come around quite often in your machine learning training.

Use the following command to create an empirical Gaussian:

```
>>> from scipy import stats
>>> emp_dist = stats.multivariate_normal(
...             kmeans.cluster_centers_.ravel())
>>> lowest_prob_idx =
...     np.argsort(emp_dist.pdf(X))[:5]
>>> np.all(X[sorted_idx] == X[lowest_prob_idx])
True
```

Using k-NN for regression

Regression is covered elsewhere in the book, but we might also want to run a regression on "pockets" of the feature space. We can think that our dataset is subject to several data processes. If this is true, only training on similar data points is a good idea.

Getting ready

Our old friend, regression, can be used in the context of clustering. Regression is obviously a supervised technique, so we'll use **k-Nearest Neighbors (k-NN)** clustering rather than KMeans.

For the k-NN regression, we'll use the K closest points in the feature space to build the regression rather than using the entire space as in regular regression.

How to do it...

For this recipe, we'll use the `iris` dataset. If we want to predict something such as the petal width for each flower, clustering by iris species can potentially give us better results. The k-NN regression won't cluster by the species, but we'll work under the assumption that the Xs will be close for the same species, or in this case, the petal length.

We'll use the `iris` dataset for this recipe:

```
>>> from sklearn import datasets  
>>> iris = datasets.load_iris()  
>>> iris.feature_names  
['sepal length (cm)', 'sepal width (cm)', 'petal  
length (cm)',  
'petal width (cm)']
```

We'll try to predict the petal length based on the sepal length and width. We'll also fit a regular linear regression to see how well the k-NN regression does in comparison:

```
>>> from sklearn.linear_model import  
LinearRegression  
>>> lr = LinearRegression()  
>>> lr.fit(X, y)  
>>> print "The MSE is: {:.2f}".format(np.power(y  
- lr.predict(X),  
2).mean())  
The MSE is: 0.15
```

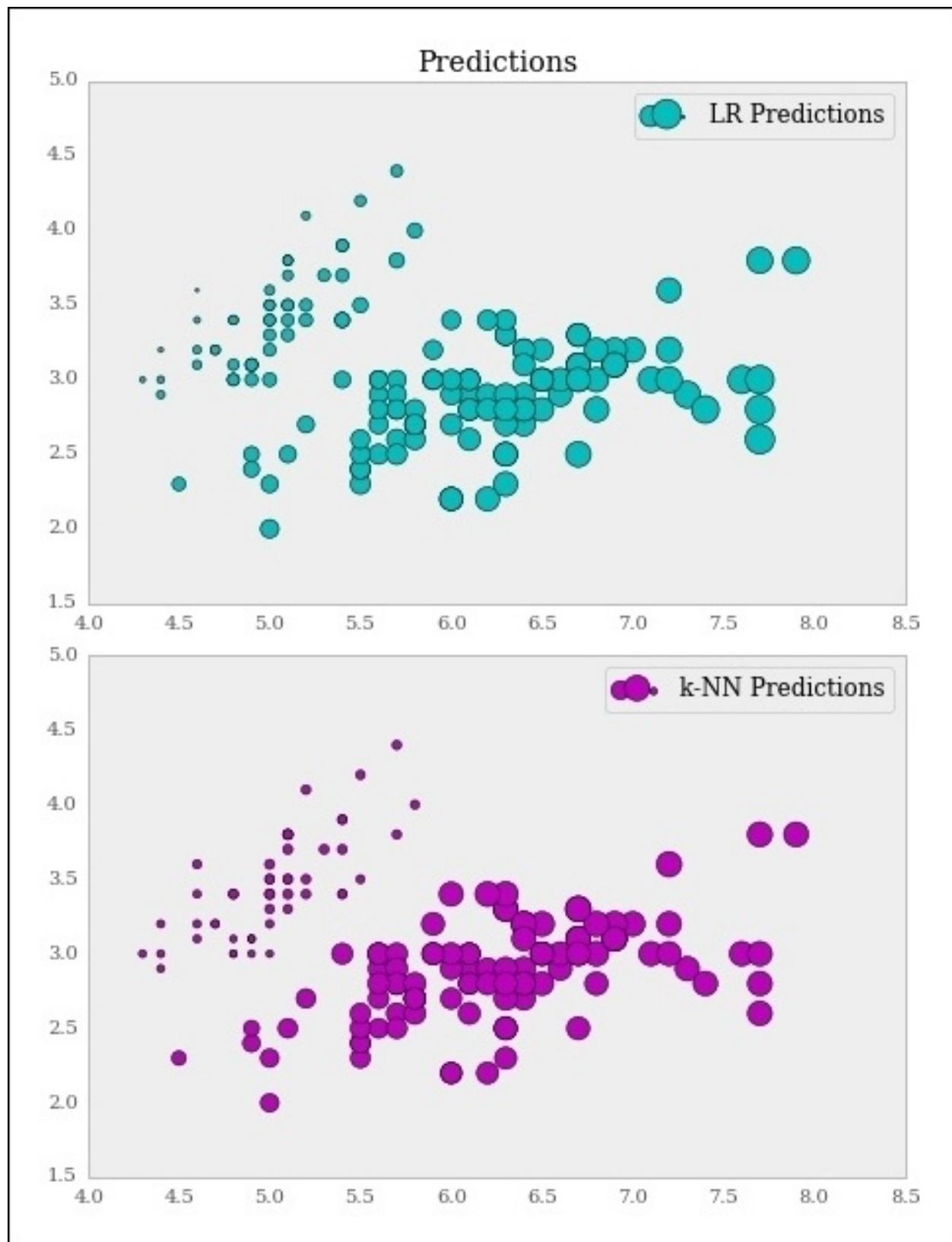
Now, for the k-NN regression, use the following code:

```
>>> from sklearn.neighbors import  
KNeighborsRegressor  
>>> knnr = KNeighborsRegressor(n_neighbors=10)  
>>> knnr.fit(X, y)  
>>> print "The MSE is: {:.2f}".format(np.power(y  
- knnr.predict(X),  
2).mean())  
The MSE is: 0.069
```

Let's look at what the k-NN regression does when we tell it to use the closest 10 points for regression:

```
>>> f, ax = plt.subplots(nrows=2, figsize=(7,  
10))  
  
>>> ax[0].set_title("Predictions")  
  
>>> ax[0].scatter(X[:, 0], X[:, 1],  
s=lr.predict(X)*80, label='LR  
Predictions', color='c', edgecolors='black')  
>>> ax[1].scatter(X[:, 0], X[:, 1],  
s=knnr.predict(X)*80, label='k-NN  
Predictions', color='m', edgecolors='black')  
  
>>> ax[0].legend()  
>>> ax[1].legend()
```

The following is the output:



It might be completely clear that the predictions are close for the most part, but let's look at the predictions for the Setosa species as compared to the actuals:

```
>>> setosa_idx =
np.where(iris.target_names=='setosa')
>>> setosa_mask = iris.target == setosa_idx[0]
>>> y[setosa_mask][:5]
array([ 0.2,  0.2,  0.2,  0.2,  0.2])
>>> knnr.predict(X)[setosa_mask][:5]
array([ 0.28,  0.17,  0.21,  0.2 ,  0.31])

>>> lr.predict(X)[setosa_mask][:5]
array([ 0.44636645,  0.53893889,  0.29846368,
0.27338255,  0.32612885])
```

Looking at the plots again, the Setosa species (upper-left cluster) is largely overestimated by linear regression, and k-NN is fairly close to the actual values.

How it works...

The k-NN regression is very simply calculated taking the average of the k closest point to the point being tested.

Let's manually predict a single point:

```
>>> example_point = x[0]
```

Now, we need to get the 10 closest points to our example_point:

```
>>> from sklearn.metrics import pairwise
>>> distances_to_example =
pairwise.pairwise_distances(X) [0]
>>> ten_closest_points =
x[np.argsort(distances_to_example) ] [:10]
>>> ten_closest_y =
y[np.argsort(distances_to_example) ] [:10]

>>> ten_closest_y.mean()
0.28000
```

We can see that this is very close to what was expected.

Chapter 4. Classifying Data with scikit-learn

This chapter will cover the following topics:

- Doing basic classifications with Decision Trees
- Tuning a Decision Tree model
- Using many Decisions Trees – random forests
- Tuning a random forest model
- Classifying data with support vector machines
- Generalizing with multiclass classification
- Using LDA for classification
- Working with QDA – a nonlinear LDA
- Using Stochastic Gradient Descent for classification
- Classifying documents with Naïve Bayes
- Label propagation with semi-supervised learning

Introduction

Classification can be very important in a lot of contexts. For example, if we want to automate some decision-making process, we can utilize classification. In cases where we need to investigate a fraud, there are so many transactions that it is impractical for a person to check all of them. Therefore, we can automate such decisions with classification.

Doing basic classifications with Decision Trees

In this recipe, we will perform basic classifications using Decision Trees. These are very nice models because they are easily understandable, and once trained in, scoring is very simple. Often, SQL statements can be used, which means that the outcome can be used by a lot of people.

Getting ready

In this recipe, we'll look at Decision Trees. I like to think of Decision Trees as the base class from which a large number of other classification methods are derived. It's a pretty simple idea that works well in a bunch of situations.

First, let's get some classification data that we can practice on:

```
>>> from sklearn import datasets  
>>> X, y =  
datasets.make_classification(n_samples=1000,  
n_features=3,  
  
n_redundant=0)
```

How to do it...

Working with Decision Trees is easy. We first need to import the object, and then fit the model:

```
>>> from sklearn.tree import  
DecisionTreeClassifier  
>>> dt = DecisionTreeClassifier()  
>>> dt.fit(X, y)  
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')  
  
>>> preds = dt.predict(X)  
>>> (y == preds).mean()  
1.0
```

As you can see, we guessed it right. Clearly, this was just a dry run, now let's investigate some of our options.

First, if you look at the `dt` object, it has several keyword arguments that determine how the object will behave. How we choose the object is important, so we'll look at

the object's effects in detail.

The first detail we'll look at is `max_depth`. This is an important parameter. It determines how many branches are allowed. This is important because a Decision Tree can have a hard time generalizing out-of-sampled data with some sort of regularization. Later, we'll see how we can use several shallow Decision Trees to make a better learner. Let's create a more complex dataset and see what happens when we allow different `max_depth`. We'll use this dataset for the rest of the recipe:

```
>>> n_features=200
>>> X, y = datasets.make_classification(750,
n_features,
n_informative=5)
>>> import numpy as np
>>> training = np.random.choice([True, False],
p=[.75, .25],
size=len(y))

>>> accuracies = []

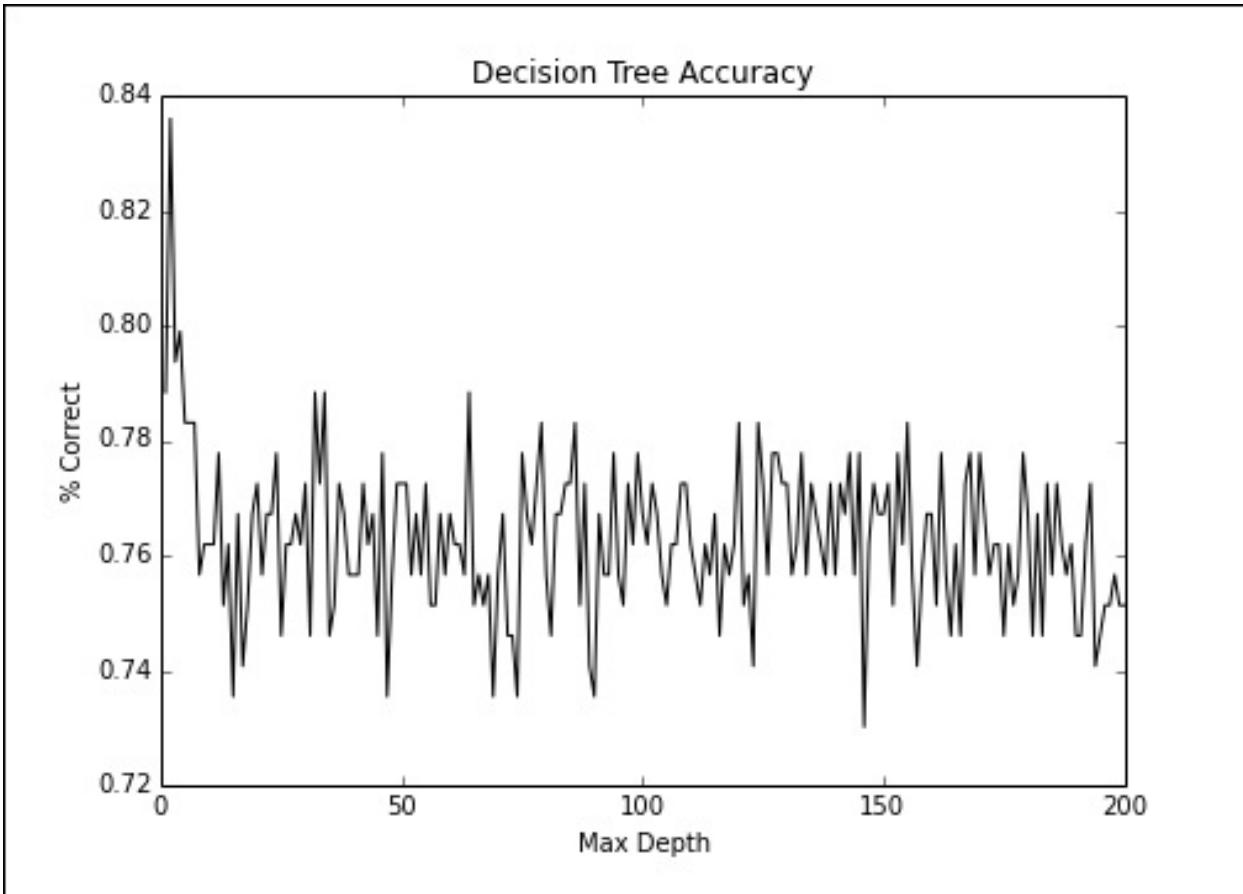
>>> for x in np.arange(1, n_features+1):
>>> dt = DecisionTreeClassifier(max_depth=x)

>>> dt.fit(X[training], y[training])

>>> preds = dt.predict(X[~training])
```

```
>>> accuracies.append( (preds ==  
y[~training]) .mean() )  
  
>>> import matplotlib.pyplot as plt  
  
>>> f, ax = plt.subplots(figsize=(7, 5))  
  
>>> ax.plot(range(1, n_features+1), accuracies,  
color='k')  
  
>>> ax.set_title("Decision Tree Accuracy")  
>>> ax.set_ylabel("% Correct")  
>>> ax.set_xlabel("Max Depth")
```

The following is the output:



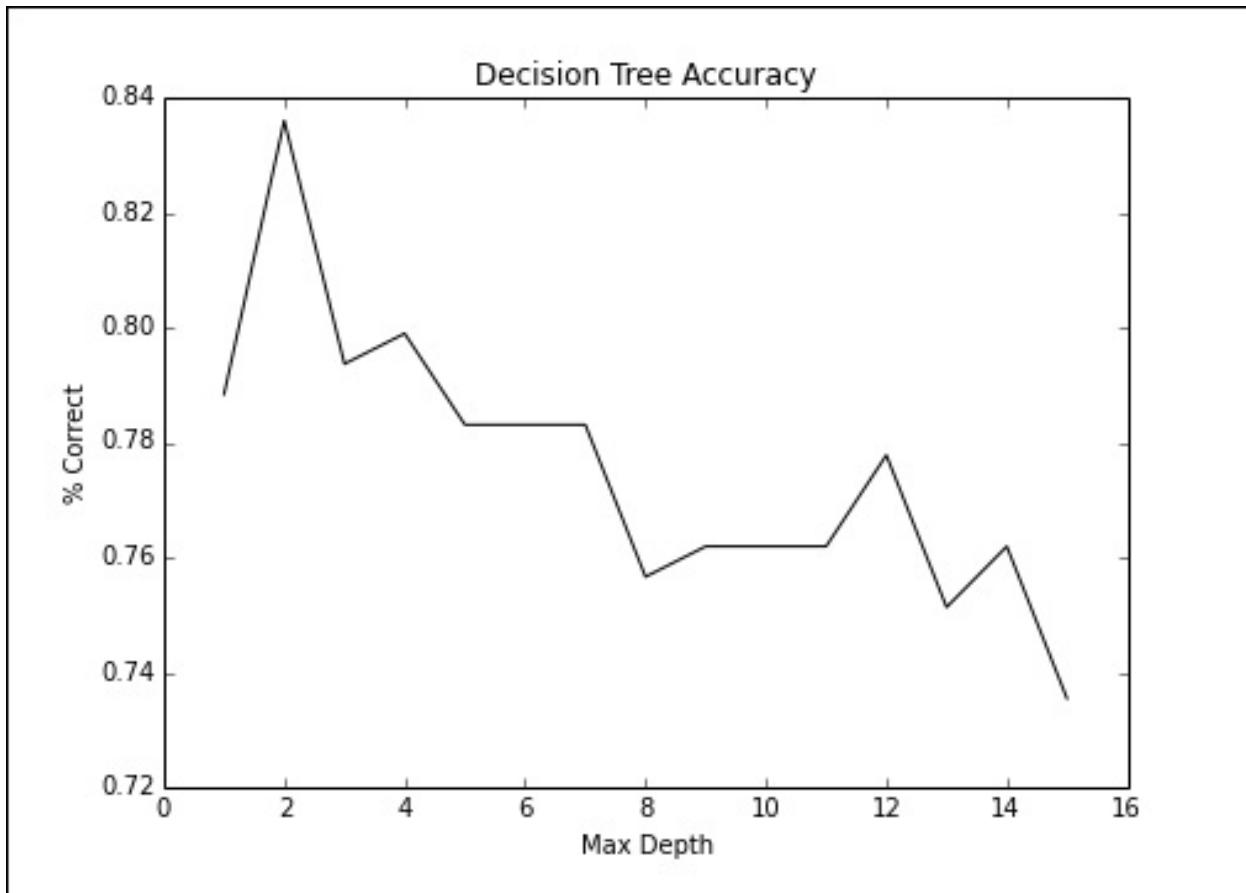
We can see that we actually get pretty accurate at a low max depth. Let's take a closer look at the accuracy at low levels, say the first 15:

```
>>> N = 15
>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.plot(range(1, n_features+1) [:N] ,
    accuracies[:N] , color='k')
```

```
>>> ax.set_title("Decision Tree Accuracy")
>>> ax.set_ylabel("% Correct")
>>> ax.set_xlabel("Max Depth")
```

The following is the output:

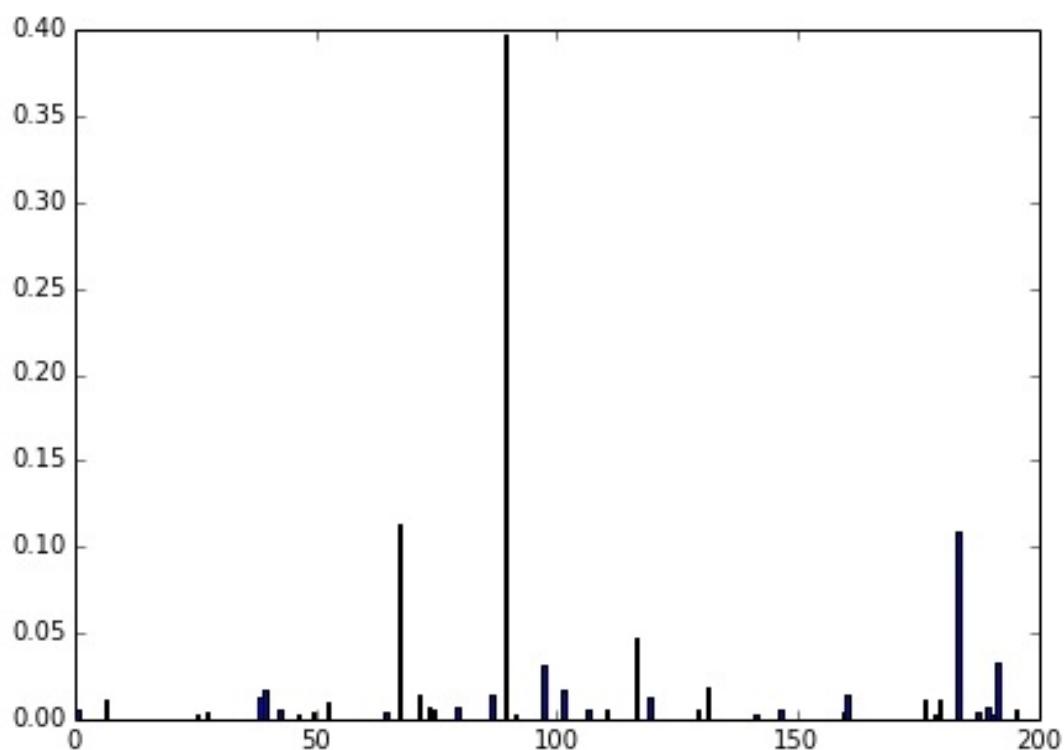


There's the spike we saw earlier; it's quite amazing to see the quick drop though. It's more likely that **Max Depth** of 1 through 3 is fairly equivalent. Decision Trees are quite good at separating rules, but they need to be reigned in.

We'll look at the `compute_importances` parameter here. It actually has a bit of a broader meaning for random forests, but we'll get acquainted with it. It's also worth noting that if you're using Version 0.16 or earlier, you will get this for free:

```
>>> dt_ci =  
DecisionTreeClassifier(compute_importances=True)  
>>> dt.fit(X, y)  
  
#plot the importances  
>>> ne0 = dt.feature_importances_ != 0  
  
>>> y_comp = dt.feature_importances_[ne0]  
>>> x_comp =  
np.arange(len(dt.feature_importances_))[ne0]  
  
>>> import matplotlib.pyplot as plt  
  
>>> f, ax = plt.subplots(figsize=(7, 5))  
>>> ax.bar(x_comp, y_comp)
```

The following is the output:



Note

Please note that you may get an error letting you know you'll no longer need to explicitly set compute importances.

As we can see, one of the features is by far the most important; several other features will follow up.

How it works...

In the simplest sense, we construct Decision Trees all the time. When thinking through situations and assigning probabilities to outcomes, we construct Decision Trees. Our rules are much more complex and involve a lot of context, but with Decision Trees, all we care about is the difference between outcomes, given that some information is already known about a feature.

Now, let's discuss the differences between **entropy** and **Gini impurity**.

Entropy is more than just the entropy value at any given variable; it states what the change in entropy is if we know an element's value. This is called **Information Gain (IG)**; mathematically it looks like the following:

$$IG(\text{Data, Known Features}) = H(\text{Data}) - H(\text{Data|Known Features})$$

For **Gini impurity**, we care about how likely one of the data points will be mislabeled given the new information.

Both entropy and Gini impurity have pros and cons; this said, if you see major differences in the working of

entropy and Gini impurity, it will probably be a good idea to re-examine your assumptions.

Tuning a Decision Tree model

If we use just the basic implementation of a Decision Tree, it will probably not fit very well. Therefore, we need to tweak the parameters in order to get a good fit. This is very easy and won't require much effort.

Getting ready

In this recipe, we will take an in-depth look at what it takes to tune a Decision Tree classifier. There are several options, and in the previous recipe, we only looked at one of these options.

We'll fit a basic model and actually look at what the Decision Tree looks like. Then, we'll re-examine after each decision and point out how various changes have influenced the structure.

If you want to follow along in this recipe, you'll need to install **pydot**.

How to do it...

Decision Trees have a lot more "knobs" when compared to most other algorithms, because of which it's easier to see what happens when we turn the knobs:

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification(1000,
20, n_informative=3)

>>> from sklearn.tree import
DecisionTreeClassifier
>>> dt = DecisionTreeClassifier()
>>> dt.fit(X, y)
```

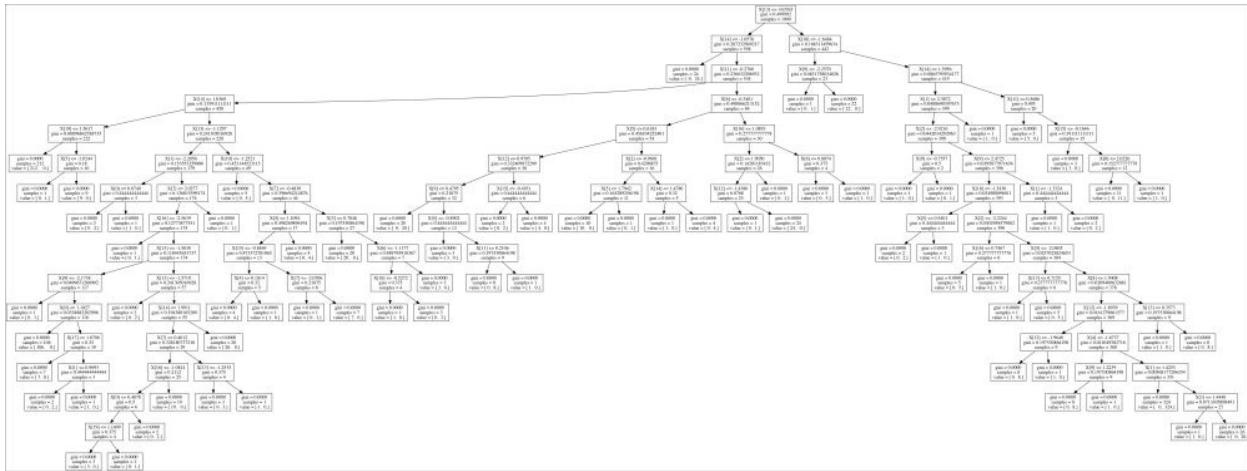
Ok, so now that we have a basic classifier fit, we can view it quite simply:

```
>>> from StringIO import StringIO
>>> from sklearn import tree
>>> import pydot

>>> str_buffer = StringIO()
>>> tree.export_graphviz(dt,
out_file=str_buffer)
>>> graph =
pydot.graph_from_dot_data(str_buffer.getvalue())
>>> graph.write("myfile.jpg")
```

The graph is almost certainly illegible, but hopefully this illustrates the complex trees that can be generated as a

result of using an unoptimized decision tree:



Wow! This is a very complex tree. It will most likely overfit the data. First, let's reduce the max depth value:

```
>>> dt = DecisionTreeClassifier(max_depth=5)
>>> dt.fit(X, y);
```

As an aside, if you're wondering why the semicolon, the `repr` by default, is seen, it is actually the model for a Decision Tree. For example, the `fit` function actually returns the Decision Tree object that allows chaining:

```
>>> dt =
DecisionTreeClassifier(max_depth=5).fit(X, y)
```

Now, let's get back to the regularly scheduled program.

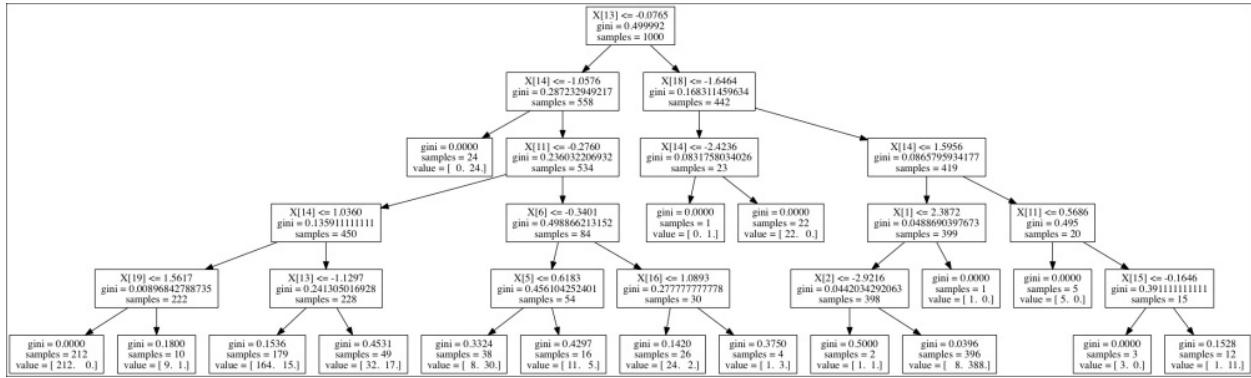
As we will plot this a few times, let's create a function:

```
>>> def plot_dt(model, filename):
        str_buffer = StringIO()
>>> tree.export_graphviz(model,
out_file=str_buffer)

>>> graph =
pydot.graph_from_dot_data(str_buffer.getvalue())
>>> graph.write_jpg(filename)

>>> plot_dt(dt, "myfile.png")
```

The following is the graph that will be generated:

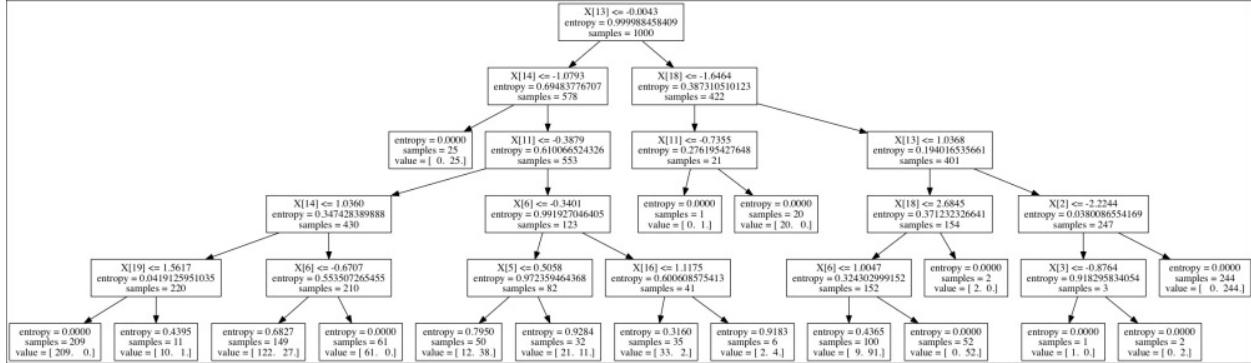


This is a much simpler tree. Let's look at what happens when we use entropy as the splitting criteria:

```
>>> dt =
DecisionTreeClassifier(criterion='entropy',
max_depth=5).fit(X, y)
```

```
>>> plot(dt, "entropy.png")
```

The following is the graph that can be generated:



It's good to see that the first two splits are the same features, and the first few after this are interspersed with similar amounts. This is a good sanity check.

Also, note how entropy for the first split is 0.999, but for the first split when using the Gini impurity is 0.5. This has to do with how different the two measures of the split of a Decision Tree are. See the following *How it works...* section for more information. However, if we want to create a Decision Tree with entropy, we must use the following command:

```
>>> dt =
DecisionTreeClassifier(min_samples_leaf=10,
criterion='entropy',
```

```
max_depth=5) .fit(X, y)
```

How it works...

Decision Trees, in general, suffer from overfitting. Quite often, left to its own devices, a Decision Tree model will overfit, and therefore, we need to think about how best to avoid overfitting; this is done to avoid complexity. A simple model will more often work better in practice than not.

We're about to see this very idea in practice. random forests will build on this idea of simple models.

Using many Decision Trees – random forests

In this recipe, we'll use random forests for classification tasks. random forests are used because they're very robust to overfitting and perform well in a variety of situations.

Getting ready

We'll explore this more in the *How it works...* section of this recipe, but random forests work by constructing a lot of very shallow trees, and then taking a vote of the class that each tree "voted" for. This idea is very powerful in machine learning. If we recognize that a simple trained classifier might only be 60 percent accurate, we can train lots of classifiers that are generally right and can then use the learners together.

How to do it...

The mechanics of training a random forest classifier is very easy with scikit-learn. In this section, we'll do the following:

1. Create a sample dataset to practice with.
2. Train a basic random forest object.
3. Take a look at some of the attributes of a trained object.

In the next recipe, we'll look at how to tune the random forest classifier. Let's start by importing datasets:

```
>>> from sklearn import datasets
```

Then, create the dataset with 1,000 samples:

```
>>> x, y = datasets.make_classification(1000)
```

Now that we have the data, we can create a classifier object and train it:

```
>>> from sklearn.ensemble import  
RandomForestClassifier  
  
>>> rf = RandomForestClassifier()  
  
>>> rf.fit(x, y)
```

The first thing we want to do is see how well we fit the training data. We can use the `predict` method for these projections:

```
>>> print "Accuracy:\t", (y ==  
rf.predict(X)).mean()  
Accuracy: 0.993  
  
>>> print "Total Correct:\t", (y ==  
rf.predict(X)).sum()  
Total Correct: 993
```

Now, let's look at some attributes and methods.

First, we'll look at some of the useful attributes; in this case, since we used defaults, they'll be the object defaults:

- `rf.criterion`: This is the criterion for how the splits are determined. The default is `gini`.
- `rf.bootstrap`: A Boolean that indicates whether we used bootstrap samples when training random forest.
- `rf.n_jobs`: The number of jobs to train and predict. If you want to use all the processors, set this to `-1`. Keep in mind that if your dataset isn't very big, it often leads to more overhead in using multiple jobs due to the data having to be serialized and moved in between processes.
- `rf.max_features`: This denotes the number of features to consider when making the best split. This

will come in handy during the tuning process.

- `rf.compute_importances`: This helps us decide whether to compute the importance of the features. See the *There's more...* section of this recipe for information on how to use this.
- `rf.max_depth`: This denotes how deep each tree can go.

There are more attributes to note; check out the official documentation for more details.

The `predict` method isn't the only useful one. We can also get the probabilities of each class from individual samples. This can be a useful feature to understand the uncertainty in each prediction. For instance, we can predict the probabilities of each sample for the various classes:

```
>>> probs = rf.predict_proba(X)

>>> import pandas as pd

>>> probs_df = pd.DataFrame(probs, columns=['0',
'1'])

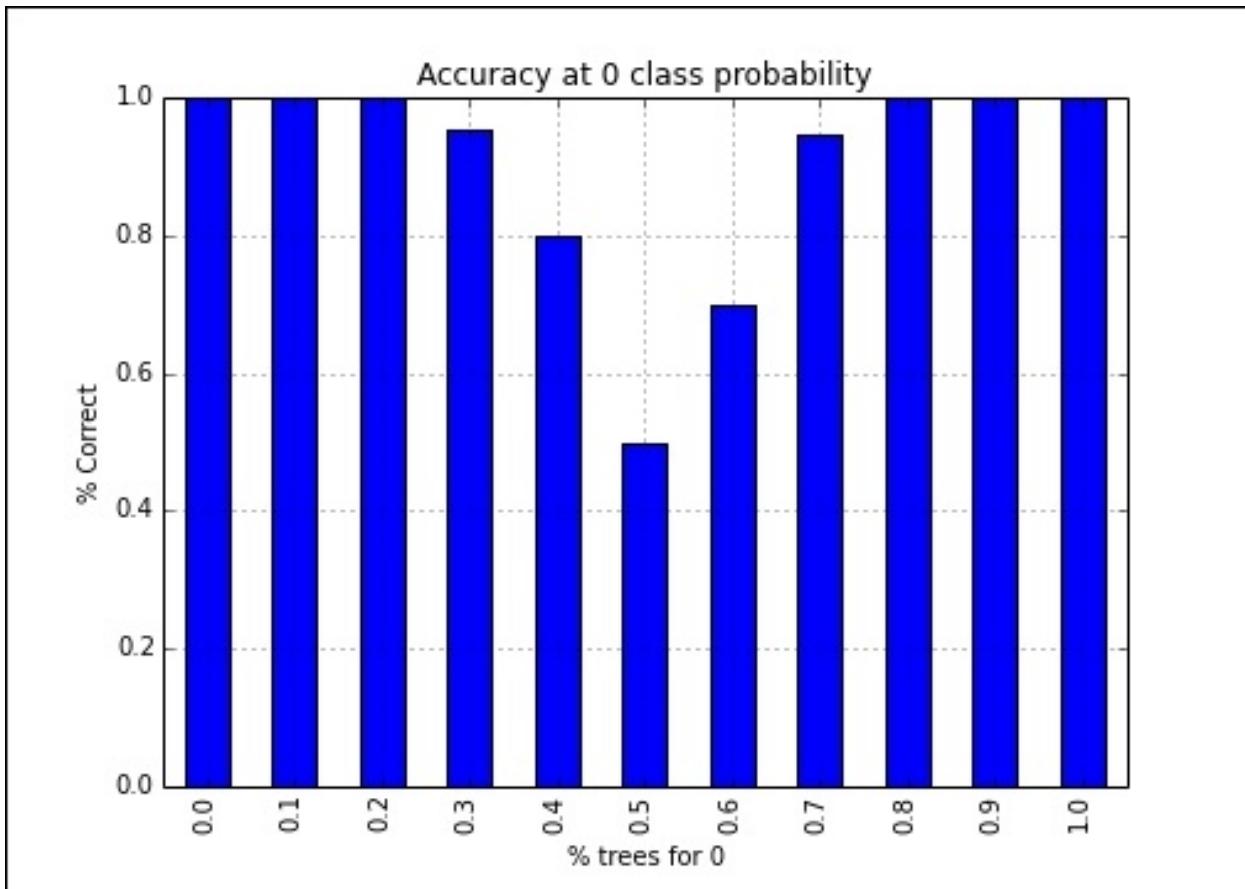
>>> probs_df['was_correct'] = rf.predict(X) == y

>>> import matplotlib.pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))
```

```
>>>  
probs_df.groupby('0').was_correct.mean().plot(ki  
nd='bar', ax=ax)  
>>> ax.set_title("Accuracy at 0 class  
probability")  
>>> ax.set_ylabel("% Correct")  
>>> ax.set_xlabel("% trees for 0")
```

The following is the output:



How it works...

Random forest works by using a predetermined number of weak Decision Trees and by training each one of these trees on a subset of data. This is critical in avoiding overfitting. This is also the reason for the `bootstrap` parameter. We have each tree trained with the following:

- The class with the most votes
- The output, if we use regression trees

There are, of course, performance considerations, which we'll cover in the next recipe, but for the purposes of understanding how random forests work, we train a bunch of average trees and get a fairly good classifier as a result.

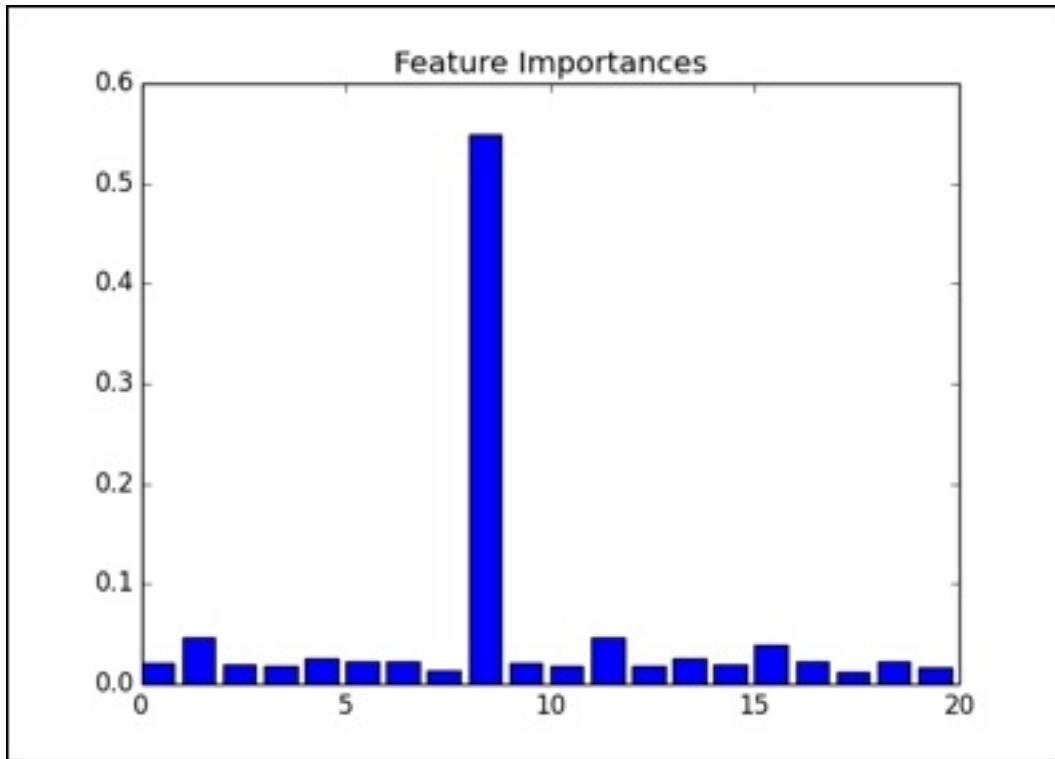
There's more...

Feature importance is a good by-product of random forests. This often helps to answer the question: *If we have 10 features, which features are most important in determining the true class of the data point?* The real-world applications are hopefully easy to see. For example, if a transaction is fraudulent, we probably want to know if there are certain signals that can be used to figure out a transaction's class more quickly.

If we want to calculate the feature importance, we need to state it when we create the object. If you use scikit-learn 0.15, you might get a warning that it is not required; in Version 0.16, the warning will be removed:

```
>>> rf =  
RandomForestClassifier(compute_importances=True)  
>>> rf.fit(X, y)  
>>> f, ax = plt.subplots(figsize=(7, 5))  
>>> ax.bar(range(len(rf.feature_importances_)),  
         rf.feature_importances_)  
>>> ax.set_title("Feature Importances")
```

The following is the output:



As we can see, certain features are much more important than others when determining if the outcome was of class 0 or class 1.

Tuning a random forest model

In the previous recipe, we reviewed how to use the random forest classifier. In this recipe, we'll walk through how to tune its performance by tuning its parameters.

Getting ready

In order to tune a random forest model, we'll need to first create a dataset that's a little more difficult to predict.

Then, we'll alter the parameters and do some preprocessing to fit the dataset better.

So, let's create the dataset first:

```
>>> from sklearn import datasets  
>>> X, y =  
datasets.make_classification(n_samples=10000,  
  
n_features=20,  
  
n_informative=15,  
  
flip_y=.5, weights=[.2, .8])
```

How to do it...

In this recipe, we will do the following:

1. Create a training and test set. We won't just sail through this recipe like we did in the previous recipe. It's an empty deed to tune a model without comparing it to a training set.
2. Fit a baseline random forest to evaluate how well we do with a naive algorithm.
3. Alter some parameters in a systematic way, and then observe what happens to the fit.

Ok, start an interpreter and import NumPy:

```
>>> import numpy as np
>>> training = np.random.choice([True, False],
p=[.8, .2],
size=y.shape)

>>> from sklearn.ensemble import
RandomForestClassifier

>>> rf = RandomForestClassifier()
>>> rf.fit(X[training], y[training])

>>> preds = rf.predict(X[~training])

>>> print "Accuracy:\t", (preds ==
y[~training]).mean()
```

```
Accuracy: 0.652239557121
```

I'm going to cheat a little bit and introduce one of the model evaluation metrics we will talk about later in the book. Accuracy is a good first metric, but using a confusion matrix will help us understand what's going on.

Let's iterate through the recommended choices for `max_features` and see what it does to the fit. We'll also iterate through a couple of `floats`, which are the fraction of the features that will be used. Use the following commands to do so:

```
>>> from sklearn.metrics import confusion_matrix

>>> max_feature_params = ['auto', 'sqrt',
'log2', .01, .5, .99]

>>> confusion_matrixes = {}

>>> for max_feature in max_feature_params:
    rf =
RandomForestClassifier(max_features=max_feature)
    rf.fit(X[training], y[training])

>>> confusion_matrixes[max_feature] =
confusion_matrix(y[~training])

>>> rf.predict(X[~training]).ravel()
```

Since I used the `ravel` method, our 2D confusion matrices

are now 1D.

Now, import pandas and look at the confusion matrix we just created:

```
>>> import pandas as pd

>>> confusion_df =
pd.DataFrame(confusion_matrixes)

>>> import itertools
>>> from matplotlib import pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> confusion_df.plot(kind='bar', ax=ax)

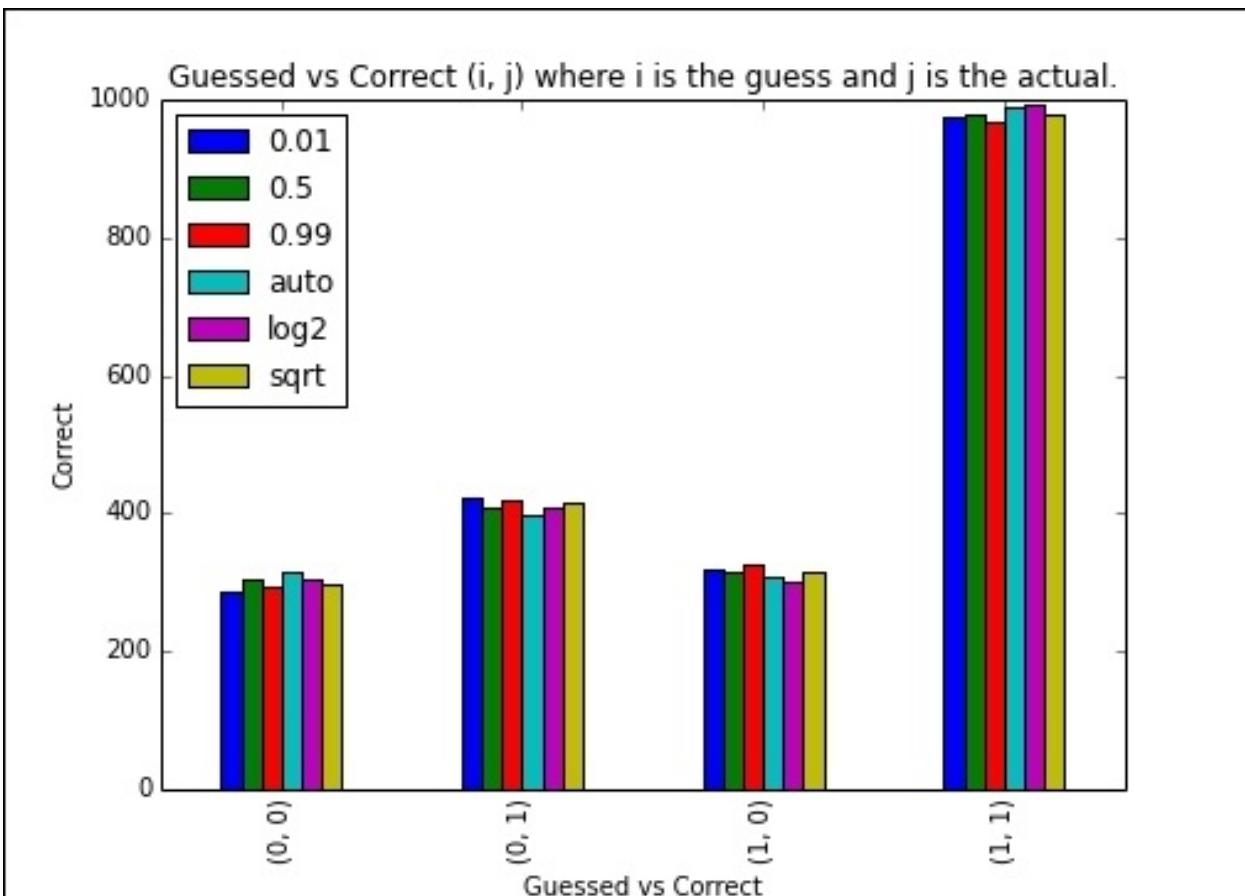
>>> ax.legend(loc='best')

>>> ax.set_title("Guessed vs Correct (i, j)
where i is the guess and j is
the actual.")

>>> ax.grid()

>>> ax.set_xticklabels([str((i, j)) for i, j in
list(itertools.product(range(2), range(2))))]);
>>> ax.set_xlabel("Guessed vs Correct")
>>> ax.set_ylabel("Correct")
```

The following is the output:



While we didn't see any real difference in performance, this is a fairly simple process to go through for your own projects. Let's try it on the choice of `n_estimator` instances, but use raw accuracy. With more than a few options, our graph is going to become very cloudy and difficult to use.

Since we're using the confusion matrix, we can get the accuracy from the trace of the confusion matrix divided

by the overall sum:

```
>>> n_estimator_params = range(1, 20)

>>> confusion_matrixes = {}

>>> for n_estimator in n_estimator_params:
    rf =
RandomForestClassifier(n_estimators=n_estimator)

    rf.fit(X[training], y[training])

    confusion_matrixes[n_estimator] =
confusion_matrix(y[~training],

rf.predict(X[~training]))
    # here's where we'll update the confusion
matrix with the
    operation we talked about

>>> accuracy = lambda x: np.trace(x) / np.sum(x,
dtype=float)
>>> confusion_matrixes[n_estimator] =
accuracy(confusion_matrixes[n_estimator])

>>> accuracy_series =
pd.Series(confusion_matrixes)

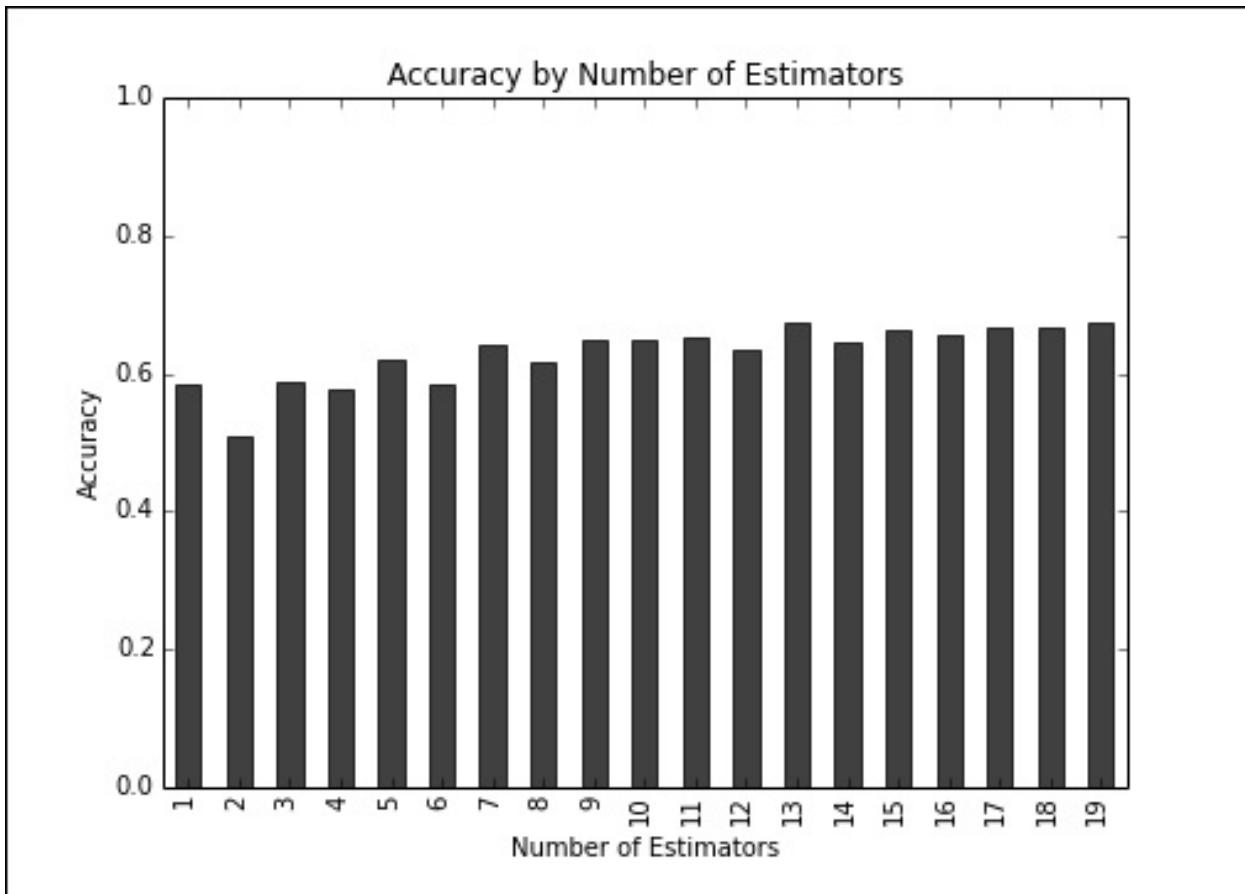
>>> import itertools
>>> from matplotlib import pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))
```

```
>>> accuracy_series.plot(kind='bar', ax=ax,
color='k', alpha=.75)
>>> ax.grid()

>>> ax.set_title("Accuracy by Number of
Estimators")
>>> ax.set_ylim(0, 1) # we want the full scope
>>> ax.set_ylabel("Accuracy")
>>> ax.set_xlabel("Number of Estimators")
```

The following is the output:



Notice how accuracy is going up for the most part. There certainly is some randomness associated with the accuracy, but the graph is up and to the right. In the following *How it works...* section, we'll talk about the association between random forest and bootstrap, and what is generally better.

How it works...

Bootstrapping is a nice technique to augment the other parts of modeling. The case often used to introduce bootstrapping is adding standard errors to a median. Here, we just estimate the outcome over and over and aggregate the estimates up to probabilities.

So, by simply increasing the number estimators, we increase the subsamples that lead to an overall faster convergence.

There's more...

We might want to speed up the training process. I alluded to this process earlier, but we can set `n_jobs` to the number of trees we want to train at the same time. This should roughly be the number of cores on the machine:

```
>>> rf = RandomForestClassifier(n_jobs=4,  
verbose=True)  
>>> rf.fit(X, y)  
[Parallel(n_jobs=4)]: Done 1 out of 4 |  
elapsed: 0.3s remaining: 0.9s  
[Parallel(n_jobs=4)]: Done 4 out of 4 |  
elapsed: 0.3s finished
```

This will also predict in parallel (verbosely):

```
>>> rf.predict(X)  
[Parallel(n_jobs=4)]: Done 1 out of 4 |  
elapsed: 0.0s remaining: 0.0s  
[Parallel(n_jobs=4)]: Done 4 out of 4 |  
elapsed: 0.0s finished  
  
array([1, 1, 0, ..., 1, 1, 1])
```

Classifying data with support vector machines

Support vector machines (SVM) is one of the techniques we will use that doesn't have an easy probabilistic interpretation. The idea behind SVMs is that we find the plane that separates the group of the dataset the "best". Here, separation means that the choice of the plane maximizes the margin between the closest points on the plane. These points are called **support vectors**.

Getting ready

SVM is one of my favorite machine learning algorithms. It was one of the first machine learning algorithms I learned in school. So, let's get some data and get started:

```
>>> from sklearn import datasets  
>>> X, y = datasets.make_classification()
```

How to do it...

The mechanics of creating a support vector classifier is very simple; there are a few options available. Therefore, we'll do the following:

1. Create an SVC object and fit it to some fake data.
2. Fit the SVC object to some example data.
3. Talk a little about the SVC options.

Import **support vector classifier (SVC)** from the support vector machine module:

```
>>> from sklearn.svm import SVC
```

```
>>> base_svm = SVC()
```

```
>>> base_svm.fit(X, y)
```

Let's look at some of the attributes:

- `c`: In cases where we don't have a well-separated set, `c` will scale the error on the margin. As `c` gets higher, the penalization for the error becomes larger and the SVM will try to find a narrow margin even if it misclassifies more points.
- `class_weight`: This denotes how much weight to give to each class in the problem. This is given as a

dictionary where classes are the keys and values are the weights associated with these classes.

- `gamma`: This is the gamma parameter for kernels and is supported by `rgb`, `sigmoid`, and `ploy`.
- `kernel`: This is the kernel to use; we'll use `linear` in the following *How it works...* section, but `rgb` is the popular and default choice.

How it works...

Like we talked about in the *Getting ready* section, SVM will try to find the plane that best bifurcates the two classes. Let's look at a simple example with two features and a well-separated outcome.

First, let's fit the dataset, and then we'll plot what's going on:

```
>>> X, y = datasets.make_blobs(n_features=2, centers=2)
>>> from sklearn.svm import LinearSVC
>>> svm = LinearSVC()
>>> svm.fit(X, y)
```

Now that we've fit the support vector machine, we'll plot its outcome at each point in the graph. This will show us the approximate decision boundary:

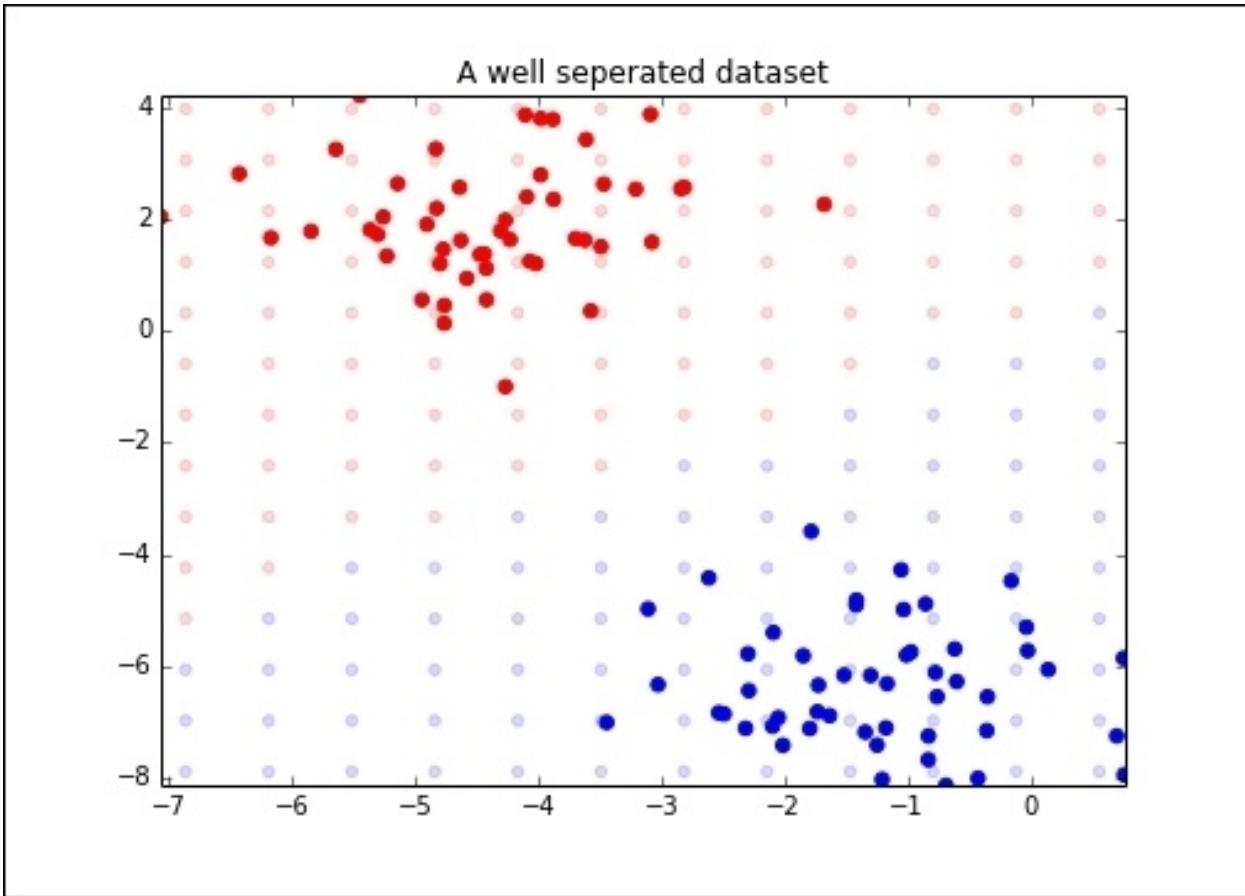
```
>>> from itertools import product
>>> from collections import namedtuple

>>> Point = namedtuple('Point', ['x', 'y',
'outcome'])
>>> decision_boundary = []
>>> xmin, xmax = np.percentile(X[:, 0], [0,
100])
>>> ymin, ymax = np.percentile(X[:, 1], [0,
100])
```

```
>>> for xpt, ypt in product(np.linspace(xmin-  
2.5, xmax+2.5, 20),  
    np.linspace(ymin-2.5, ymax+2.5, 20)):  
    p = Point(xpt, ypt, svm.predict([xpt,  
ypt]))  
    decision_boundary.append(p)

>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> import numpy as np
>>> colors = np.array(['r', 'b'])
>>> for xpt, ypt, pt in decision_boundary:
    ax.scatter(xpt, ypt, color=colors[pt[0]],  
alpha=.15)
    ax.scatter(X[:, 0], X[:, 1],  
color=colors[y], s=30)
    ax.set_ylim(ymin, ymax)
    ax.set_xlim(xmin, xmax)
    ax.set_title("A well separated dataset")
```

The following is the output:



Let's look at another example, but this time the **decision boundary** will not be so clear:

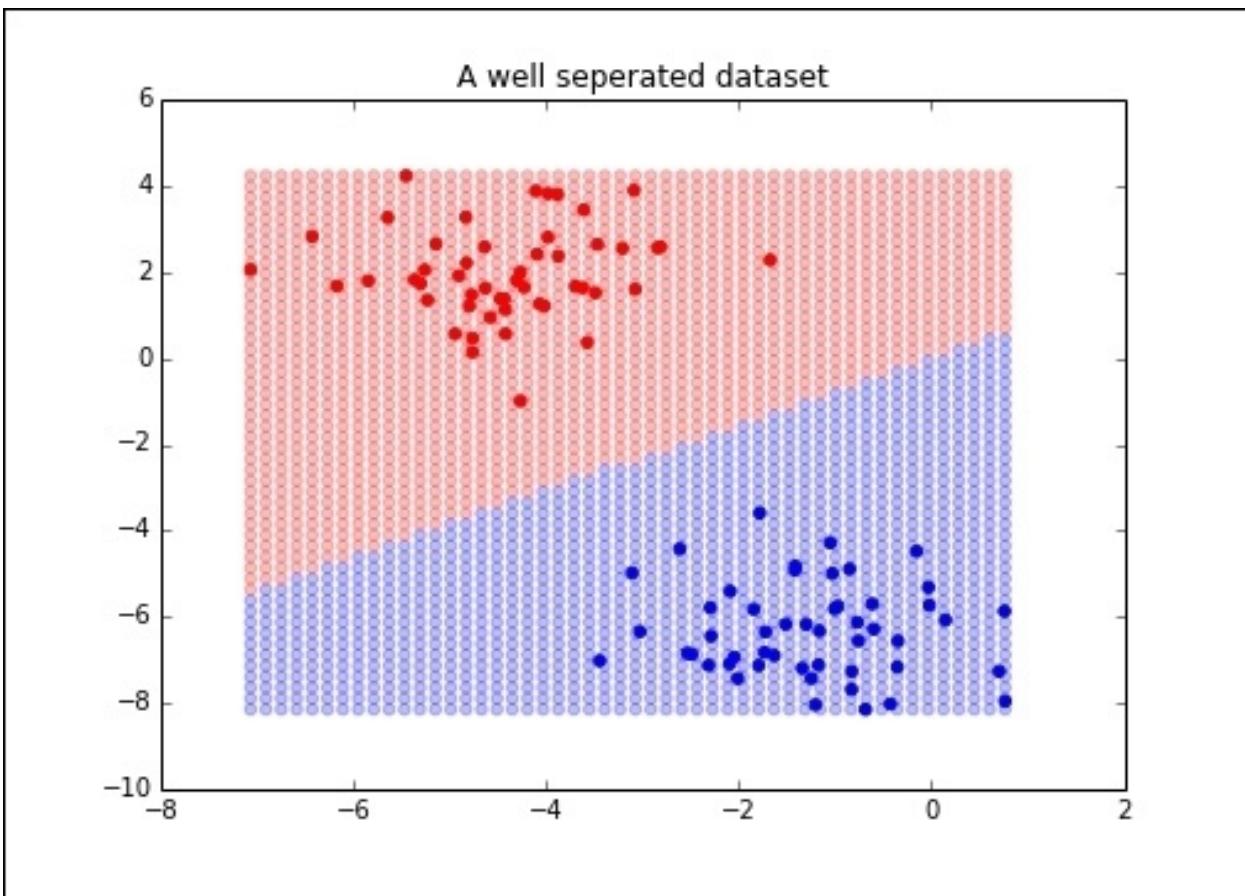
```
>>> X, y =  
datasets.make_classification(n_features=2,  
n_classes=2,  
n_informative=2,  
n_redundant=0)
```

As we can see, this is not a problem that will easily be solved by a linear classification rule.

While we will not use this in practice, let's have a look at the decision boundary. First, let's retrain the classifier with the new datapoints:

```
>>> svm.fit(X, y)
>>> xmin, xmax = np.percentile(X[:, 0], [0,
100])
>>> ymin, ymax = np.percentile(X[:, 1], [0,
100])
>>> test_points = np.array([[xx, yy] for xx, yy
in
product(np.linspace(xmin, xmax),
np.linspace(ymin,
ymax))])
>>> test_preds = svm.predict(test_points)
>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> import numpy as np
>>> colors = np.array(['r', 'b'])
>>> ax.scatter(test_points[:, 0], test_points[:, 1],
color=colors[test_preds],
alpha=.25)
>>> ax.scatter(X[:, 0], X[:, 1],
color=colors[y])
>>> ax.set_title("A well separated dataset")
```

The following is the output:



As we saw, the decision line isn't perfect, but at the end of the day, this is the best Linear SVM we will get.

There's more...

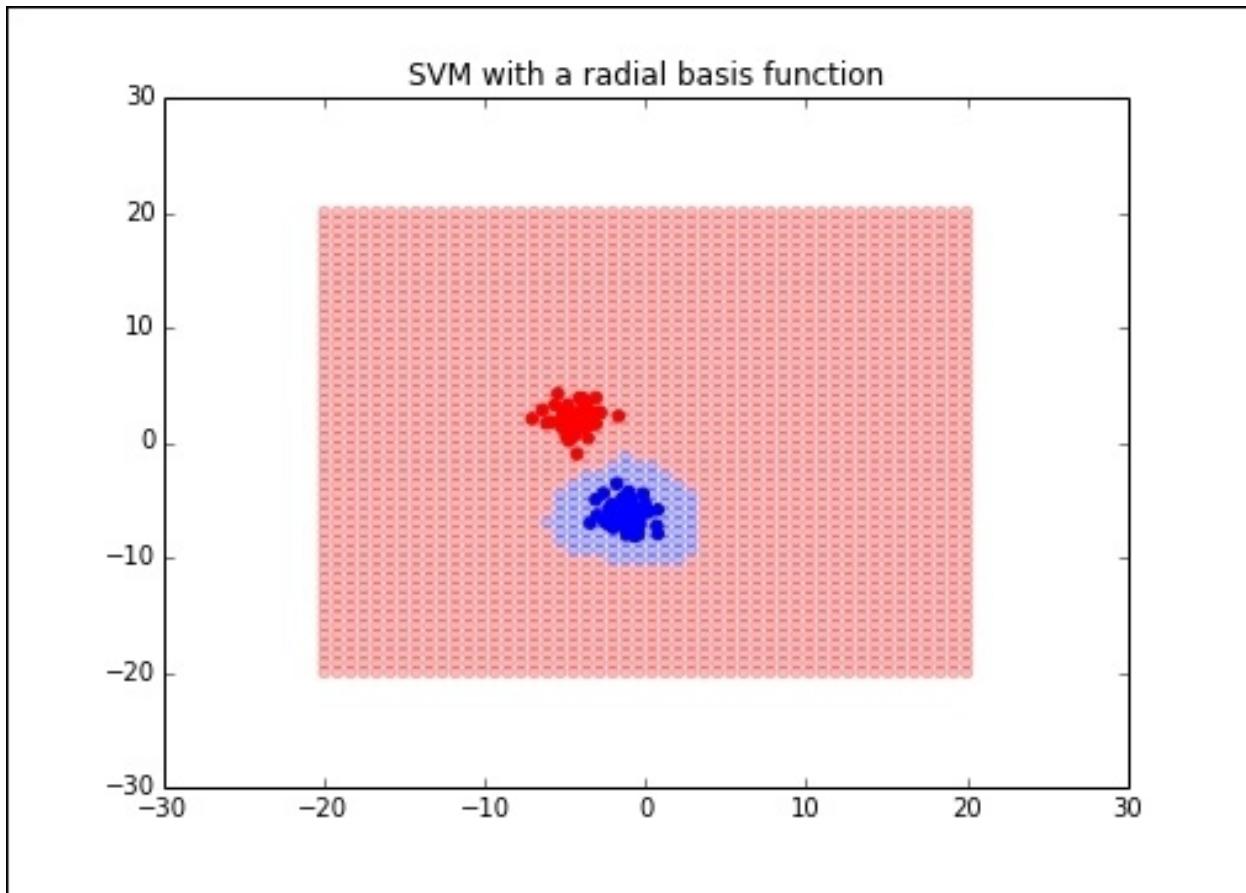
While we might not be able to get a better Linear SVM by default, the SVC classifier in scikit-learn will use the radial basis function. We've seen this function before, but let's take a look and see what it does to the decision boundaries of the dataset we just fit:

```
>>> radial_svm = SVC(kernel='rbf')
>>> radial_svm.fit(X, y)
>>> xmin, xmax = np.percentile(X[:, 0], [0,
100])
>>> ymin, ymax = np.percentile(X[:, 1], [0,
100])
>>> test_points = np.array([[xx, yy] for xx, yy
in
product(np.linspace(xmin, xmax),
np.linspace(ymin,
ymax))])
>>> test_preds = radial_svm.predict(test_points)

>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> import numpy as np
>>> colors = np.array(['r', 'b'])
>>> ax.scatter(test_points[:, 0], test_points[:, 1],
color=colors[test_preds],
alpha=.25)
>>> ax.scatter(X[:, 0], X[:, 1],
```

```
color=colors[y])  
>>> ax.set_title("SVM with a radial basis  
function")
```

The following is the output:



As we can see, the decision boundary has been altered. We can even pass in our own radial basis function, if needed:

```
>>> def test_kernel(x, y):
```

```
""" Test kernel that returns the
exponentiation of the dot of the
X and y matrices.

This looks an awful lot like the log hazards
if you're familiar with survival analysis.

"""

return np.exp(np.dot(X, y.T))
>>> test_svc = SVC(kernel=test_kernel)
>>> test_svc.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None,
coef0=0.0, degree=3,
gamma=0.0, kernel=<function test_kernel at
0x121fdfb90>,
max_iter=-1, probability=False,
random_state=None,
shrinking=True, tol=0.001, verbose=False)
```

Generalizing with multiclass classification

In this recipe, we'll look at multiclass classification. Depending on your choice of algorithm, you either get multiclass classification for free, or you have to define a scheme for comparison.

Getting ready

When working with linear models such as logistic regression, we need to use `OneVsRestClassifier`. This scheme will create a classifier for each class.

How to do it...

First, we'll walk through a cursory example of a Decision Tree fitting a multiclass dataset. Like we discussed earlier, we get multiclass for free with some classifiers, so we'll just fit the example to prove that it works, and move on.

Second, we'll actually incorporate `OneVsRestClassifier` into our model:

```
>>> from sklearn import datasets
>>> X, y =
datasets.make_classification(n_samples=10000,
n_classes=3,
n_informative=3)

>>> from sklearn.tree import
DecisionTreeClassifier
>>> dt = DecisionTreeClassifier()
>>> dt.fit(X, y)
>>> dt.predict(X)
array([1, 1, 0, ..., 2, 1, 1])
```

As you can see, we were able to fit a classifier with minimum effort.

Now, let's move on to the case of the multiclass classifier. This will require us to import `OneVsRestClassifier`.

We'll also import LogisticRegression while we're at it:

```
>>> from sklearn.multiclass import  
OneVsRestClassifier  
>>> from sklearn.linear_model import  
LogisticRegression
```

Now, we'll override the LogisticRegression classifier. Also, notice that we can parallelize this. If we think about how OneVsRestClassifier works, it's just training separate models and then comparing them. So, we can train the data separately at the same time:

```
>>> mlr =  
OneVsRestClassifier(LogisticRegression(),  
n_jobs=2)  
>>> mlr.fit(X, y)  
>>> mlr.predict(X)  
array([1, 1, 0, ..., 2, 1, 1])
```

How it works...

If we want to quickly create our own OneVsRestClassifier, how might we do it?

First, we need to construct a way to iterate through the classes and train a classifier for each classifier. Then, we need to predict each class first:

```
>>> import numpy as np
>>> def train_one_vs_rest(y, class_label):
    y_train = (y == class_label).astype(int)
    return y_train

>>> classifiers = []
>>> for class_i in sorted(np.unique(y)):
    l = LogisticRegression()
    y_train = train_one_vs_rest(y, class_i)
    l.fit(X, y_train)
    classifiers.append(l)
```

Ok, so now that we have a one versus rest scheme set up, all we need to do is evaluate the data point's likelihood for each classifier. We will then assign the classifier to the data point with the largest likelihood.

For example, let's predict $x[0]$:

```
for classifier in classifiers
>>> print classifier.predict_proba(x[0])
```

```
[ [ 0.90443776  0.09556224] ]  
[ [ 0.03701073  0.96298927] ]  
[ [ 0.98492829  0.01507171] ]
```

As you can see, the second classifier (the one in index 1) has the highest likelihood of being "positive", therefore we'll assign 1 to this point.

Using LDA for classification

Linear Discriminant Analysis (LDA) attempts to fit a linear combination of features to predict the outcome variable. LDA is often used as a preprocessing step. We'll walk through both methods in this recipe.

Getting ready

In this recipe, we will do the following:

1. Grab stock data from Yahoo.
2. Rearrange it in a shape we're comfortable with.
3. Create an LDA object to fit and predict the class labels.
4. Give an example of how to use LDA for dimensionality reduction.

How to do it...

In this example, we will perform an analysis similar to **Altman's Z-score**. In this paper, Altman looked at a company's likelihood of defaulting within two years based on several financial metrics. The following is taken from the Wiki page of Altman's Z-score:

$T_1 = \text{Working Capital} / \text{Total Assets}$. Measures liquid assets in relation to the size of the company.

$T_2 = \text{Retained Earnings} / \text{Total Assets}$. Measures profitability that reflects the company's age and earning power.

$T_3 = \text{Earnings Before Interest and Taxes} / \text{Total Assets}$. Measures operating efficiency apart from tax and leveraging factors. It recognizes operating earnings as being important to long-term viability.

$T_4 = \text{Market Value of Equity} / \text{Book Value of Total Liabilities}$. Adds market dimension that can show up security price fluctuation as a possible red flag.

$T_5 = \text{Sales} / \text{Total Assets}$. Standard measure for total asset turnover (varies greatly from industry to industry).

From Wikipedia:

[1]: Altman, Edward I. (September 1968). ""Financial Ratios, Discriminant Analysis and the Prediction of Corporate Bankruptcy"". *Journal of Finance*: 189–209.

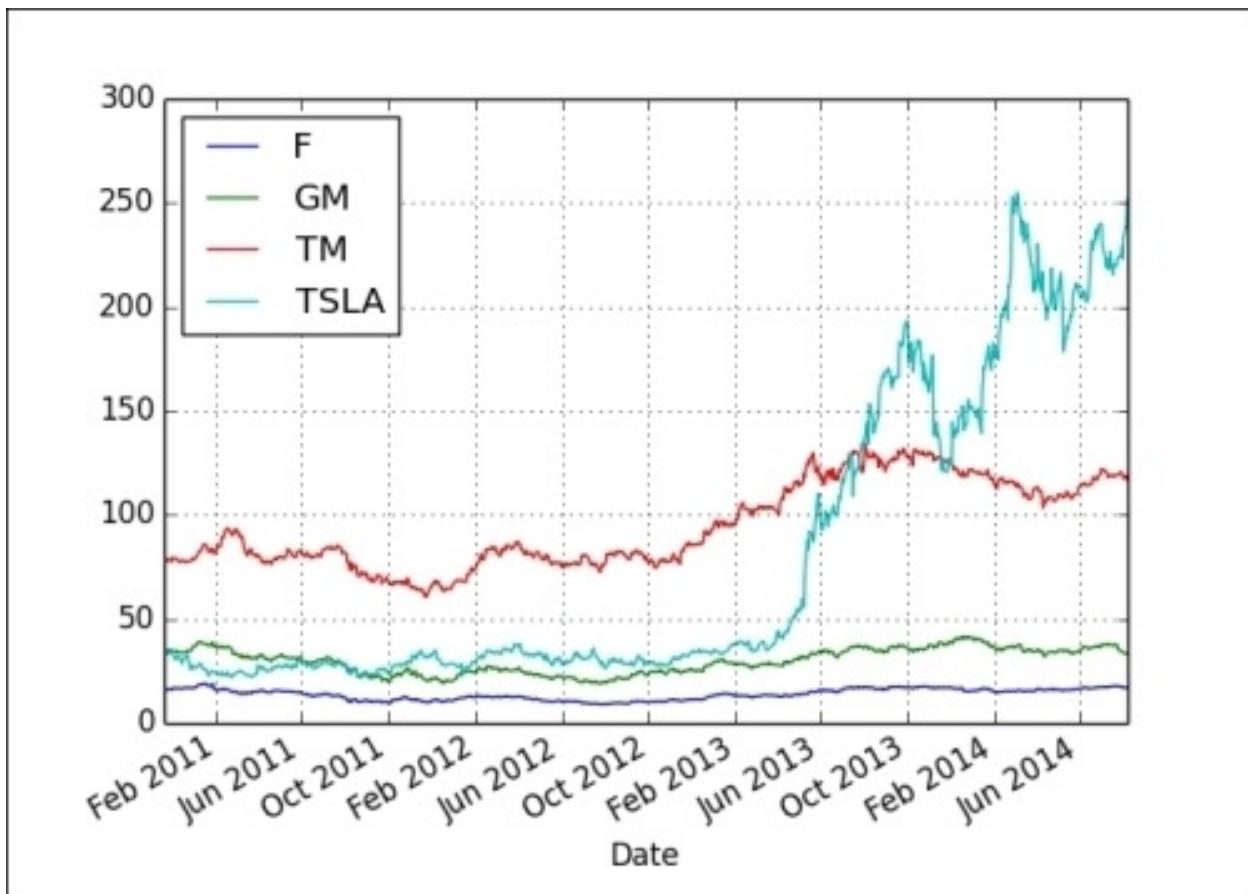
In this analysis, we'll look at some financial data from Yahoo via pandas. We'll try to predict if a stock will be higher in exactly 6 months from today, based on the current attribute of the stock. It's obviously nowhere near as refined as Altman's Z-score. Let's use a basket of auto stocks:

```
>>> tickers = ["F", "TM", "GM", "TSLA"]
>>> from pandas.io import data as external_data
>>> stock_panel =
external_data.DataReader(tickers, "yahoo")
```

This data structure is `panel` from pandas. It's similar to an OLAP cube or a 3D `DataFrame`. Let's take a look at the data to get some familiarity with closes since that's what we care about while comparing:

```
>>> stock_df = stock_panel.Close.dropna()
>>> stock_df.plot(figsize=(7, 5))
```

The following is the output:



Ok, so now we need to compare each stock price with its price in 6 months. If it's higher, we'll code it with 1, and if not, we'll code that with 0.

To do this, we'll just shift the dataframe back 180 days and compare:

```
#this dataframe indicates if the stock was
higher in 180 days
>>> classes = (stock_df.shift(-180) >
stock_df).astype(int)
```

The next thing we need to do is flatten out the dataset:

```
>>> x = stock_panel.to_frame()  
>>> classes = classes.unstack()  
>>> classes = classes.swaplevel(0,  
1).sort_index()  
>>> classes = classes.to_frame()  
>>> classes.index.names = ['Date', 'minor']  
>>> data = x.join(classes).dropna()  
>>> data.rename(columns={0: 'is_higher'},  
inplace=True)  
>>> data.head()
```

The following is the output:

		Open	High	Low	Close	Volume	Adj Close	is_higher
Date	minor							
2010-11-18	F	16.77	16.87	16.05	16.12	256937900	15.07	0
	GM	35.00	35.99	33.89	34.19	457044300	33.61	0
	TM	77.36	77.51	76.83	77.29	989100	77.29	0
	TSLA	30.67	30.74	28.92	29.89	956100	29.89	0
2010-11-19	F	16.02	16.38	15.83	16.28	130323600	15.22	0

Ok, so now we need to create matrices to SciPy. To do this, we'll use the `patsy` library. This is a great library that can be used to create a design matrix in a fashion similar to R:

```

>>> import patsy
>>> x = patsy.dmatrix("Open + High + Low + Close
+ Volume +
                           is_higher - 1",
data.reset_index(),
                           return_type='dataframe')
>>> x.head()

```

The following is the output:

	Open	High	Low	Close	Volume	is_higher
0	16.77	16.87	16.05	16.12	256937900	0
1	35.00	35.99	33.89	34.19	457044300	0
2	77.36	77.51	76.83	77.29	989100	0
3	30.67	30.74	28.92	29.89	956100	0
4	16.02	16.38	15.83	16.28	130323600	0

patsy is a very strong package, for example, suppose we want to apply some of the preprocessing from [Chapter 1](#), *Premodel Workflow*. In patsy, it's possible, like in R, to modify the formula in a way that corresponds to modifications in the design matrix. It won't be done here, but if we want to scale the value to mean 0 and standard deviation 1, the function will be "scale(open) +

```
scale(high)".
```

Awesome! So, now that we have our dataset, let's fit the LDA object:

```
>>> import pandas as pd  
>>> from sklearn.lda import LDA  
>>> lda = LDA()  
>>> lda.fit(X.ix[:, :-1], X.ix[:, -1]);
```

We can see that it's not too bad when predicting against the dataset. Certainly, we will want to improve this with other parameters and test the model:

```
>>> from sklearn.metrics import  
classification_report  
>>> print classification_report(X.ix[:,  
-1].values,  
  
lda.predict(X.ix[:, :-1]))  
precision recall f1-score  
support  
0.0 0.63 0.59 0.61  
1895  
1.0 0.60 0.64 0.62  
1833  
avg / total 0.61 0.61 0.61  
3728
```

These metrics describe how the model fits the data in various ways.

The `precision` and `recall` parameters are fairly similar. In some ways, as shown in the following list, they can be thought of as conditional proportions:

- For `precision`, given the model predicts a positive value, what proportion of this is correct?
- For `recall`, given the state of one class is true, what proportion did we "select"? I say, select because recall is a common metric in search problems. For example, there can be a set of underlying web pages that, in fact, relate to a search term—the proportion that is returned.

The `f1-score` parameter attempts to summarize the relationship between `recall` and `precision`.

How it works...

LDA is actually fairly similar to clustering that we did previously. We fit a basic model from the data. Then, once we have the model, we try to predict and compare the likelihoods of the data given in each class. We choose the option that's more likely.

LDA is actually a simplification of QDA, which we'll talk about in the next chapter. Here, we assume that the covariance of each class is the same, but in QDA, the assumption is relaxed. Think about the connections between KNN and GMM and the relationship there and here.

Working with QDA – a nonlinear LDA

QDA is the generalization of a common technique such as quadratic regression. It is simply a generalization of the model to allow for more complex models to fit, though, like all things, when allowing complexity to creep in, we make our life more difficult.

Getting ready

We will expand on the last recipe and look at **Quadratic Discernment Analysis (QDA)** via the QDA object.

We said we made an assumption about the covariance of the model. Here, we will relax the assumption.

How to do it...

QDA is aptly a member of the `qda` module. Use the following commands to use QDA:

```
>>> from sklearn.qda import QDA
>>> qda = QDA()

>>> qda.fit(X.ix[:, :-1], X.ix[:, -1])
>>> predictions = qda.predict(X.ix[:, :-1])
>>> predictions.sum()
2812.0

>>> from sklearn.metrics import
classification_report
>>> print classification_report(X.ix[:, -1].values, predictions)
          precision    recall  f1-score
support
0.0          0.75     0.36    0.49
1895
1.0          0.57     0.88    0.69
1833
avg / total       0.66     0.62    0.59
3728
```

As you can see, it's about equal on the whole. If we look back at the LDA recipe, we can see large changes as opposed to the QDA object for class 0 and minor differences for class 1.

How it works...

Like we talked about in the last recipe, we essentially compare likelihoods here. So, how do we compare likelihoods? Let's just use the price at hand to attempt to classify `is_higher`.

We'll assume that the closing price is log-normally distributed. In order to compute the likelihood for each class, we need to create the subsets of closes as well as a training and test set for each class. As a sneak peak to the next chapter, we'll use the built-in cross validation methods:

```
>>> from sklearn import cross_validation as cv

>>> import scipy.stats as sp

>>> for test, train in
cv.ShuffleSplit(len(X.Close), n_iter=1):
    train_set = X.iloc[train]
    train_close = train_set.Close

    train_0 =
train_close[~train_set.is_higher.astype(bool)]
    train_1 =
train_close[train_set.is_higher.astype(bool)]

    test_set = X.iloc[test]
    test_close = test_set.Close.values
```

```
    ll_0 = sp.norm.pdf(test_close,  
train_0.mean())  
    ll_1 = sp.norm.pdf(test_close,  
train_1.mean())
```

Now that we have likelihoods for both classes, we can compare and assign classes:

```
>>> (ll_0 > ll_1).mean()  
0.15588673621460505
```

Using Stochastic Gradient Descent for classification

As was discussed in [Chapter 2](#), *Working with Linear Models*, Stochastic Gradient Descent is a fundamental technique to fit a model for regression. There are natural connections between the two techniques, as the name so obviously implies.

Getting ready

In regression, we minimized a cost function that penalized for bad choices on a continuous scale, but for classification, we'll minimize a cost function that penalizes for two (or more) cases.

How to do it...

First, let's create some very basic data:

```
>>> from sklearn import datasets  
>>> X, y = datasets.make_classification()
```

Next, we'll create a `SGDClassifier` instance:

```
>>> from sklearn import linear_model  
>>> sgd_clf = linear_model.SGDClassifier()
```

As usual, we'll fit the model:

```
>>> sgd_clf.fit(X, y)  
SGDClassifier(alpha=0.0001, class_weight=None,  
epsilon=0.1, eta0=0.0,  
            fit_intercept=True, l1_ratio=0.15,  
            learning_rate='optimal',  
loss='hinge', n_iter=5,  
            n_jobs=1, penalty='l2',  
power_t=0.5, random_state=None,  
            shuffle=False, verbose=0,  
warm_start=False)
```

We can set the `class_weight` parameter to account for the varying amounts of unbalance in a dataset.

The Hinge loss function is defined as follows:

$$\max(0, 1 - ty)$$

Here, τ is the true classification denoted as +1 for one case and -1 for the other. The vector of coefficients is denoted by y as fit from the model, and x is the value of interest. There is also an intercept for good measure. To put it another way:

$$t \in -1, 1$$

$$y = \beta x + b$$

Classifying documents with Naïve Bayes

Naïve Bayes is a really interesting model. It's somewhat similar to k-NN in the sense that it makes some assumptions that might oversimplify reality, but still perform well in many cases.

Getting ready

In this recipe, we'll use Naïve Bayes to do document classification with sklearn. An example I have personal experience of is using the words that make up an account descriptor in accounting, such as Accounts Payable, and determining if it belongs to Income Statement, Cash Flow Statement, or Balance Sheet.

The basic idea is to use the word frequency from a labeled test corpus to learn the classifications of the documents. Then, we can turn this on a training set and attempt to predict the label.

We'll use the `newsgroups` dataset within sklearn to play with the Naïve Bayes model. It's a nontrivial amount of data, so we'll fetch it instead of loading it. We'll also limit the categories to `rec.autos` and `rec.motorcycles`:

```
>>> from sklearn.datasets import  
fetch_20newsgroups  
  
>>> categories = ["rec.autos",  
"rec.motorcycles"]  
>>> newsgroups =  
fetch_20newsgroups(categories=categories)  
  
#take a look  
>>> print "\n".join(newsgroups.data[:1])
```

```
From: gregl@zimmer.CSUFresno.EDU (Greg Lewis)
Subject: Re: WARNING.....(please read)...
Keywords: BRICK, TRUCK, DANGER
Nntp-Posting-Host: zimmer.csufresno.edu
Organization: CSU Fresno
Lines: 33
[...]
```

```
>>> newgroups.target_names[newgroups.target[:1]]
'rec.autos'
```

Now that we have `newgroups`, we'll need to represent each document as a bag of words. This representation is what gives Naïve Bayes its name. The model is "naive" because documents are classified without regard for any intradocument word covariance. This might be considered a flaw, but Naïve Bayes has been shown to work reasonably well.

We need to preprocess the data into a bag-of-words matrix. This is a sparse matrix that has entries when the word is present in the document. This matrix can become quite large, as illustrated:

```
>>> from sklearn.feature_extraction.text import
CountVectorizer

>>> count_vec = CountVectorizer()
>>> bow =
count_vec.fit_transform(newgroups.data)
```

This matrix is a sparse matrix, which is the length of the number of documents by each word. The document and word value of the matrix are the frequency of the particular term:

```
>>> bow
<1192x19177 sparse matrix of type '<type
'numpy.int64'>'
      with 164296 stored elements in Compressed
Sparse Row format>
```

We'll actually need the matrix as a dense array for the Naïve Bayes object. So, let's convert it back:

```
>>> bow = np.array(bow.todense())
```

Clearly, most of the entries are 0, but we might want to reconstruct the document counts as a sanity check:

```
>>> words =
np.array(count_vec.get_feature_names())
>>> words[bow[0] > 0][:5]
array([u'10pm', u'1qh336innf15', u'33',
u'93740',
u'_____',
''],
      dtype='<U79')
```

Now, are these the examples in the first document? Let's check that using the following command:

```
>>> '10pm' in newgroups.data[0].lower()
True
>>> 'lqh336innfl5' in newgroups.data[0].lower()
True
```

How to do it...

Ok, so it took a bit longer than normal to get the data ready, but we're dealing with text data that isn't as quickly represented as a matrix as the data we're used to.

However, now that we're ready, we'll fire up the classifier and fit our model:

```
>>> from sklearn import naive_bayes  
>>> clf = naive_bayes.GaussianNB()
```

Before we fit the model, let's split the dataset into a training and test set:

```
>>> mask = np.random.choice([True, False],  
len(bow))  
>>> clf.fit(bow[mask], newgroups.target[mask])  
>>> predictions = clf.predict(bow[~mask])
```

Now that we fit a model on a test set, and then predicted the training set in an attempt to determine which categories go with which articles, let's get a sense of the approximate accuracy:

```
>>> np.mean(predictions ==  
newgroups.target[~mask])  
0.92446043165467628
```

How it works...

The fundamental idea of how Naïve Bayes works is that we can estimate the probability of some data point being a class, given the feature vector.

This can be rearranged via the Bayes formula to give the MAP estimate for the feature vector. This MAP estimate chooses the class for which the feature vector's probability is maximized.

There's more...

We can also extend Naïve Bayes to do multiclass work. Instead of assuming a Gaussian likelihood, we'll use a multinomial likelihood.

First, let's get a third category of data:

```
>>> from sklearn.datasets import  
fetch_20newsgroups  
>>> mn_categories = ["rec.autos",  
"rec.motorcycles",  
                     "talk.politics.guns"]  
>>> mn_newsgroups =  
fetch_20newsgroups(categories=mn_categories)
```

We'll need to vectorize this just like the class case:

```
>>> mn_bow =  
count_vec.fit_transform(mn_newsgroups.data)  
>>> mn_bow = np.array(mn_bow.todense())
```

Let's create a mask array to train and test:

```
>>> mn_mask = np.random.choice([True, False],  
len(mn_newsgroups.data))  
>>> multinom = naive_bayes.MultinomialNB()  
>>> multinom.fit(mn_bow[mn_mask],  
mn_newsgroups.target[mn_mask])  
  
>>> mn_predict =
```

```
multinom.predict(mn_bow[~mn_mask])
>>> np.mean(mn_predict ==
mn_newgroups.target[~mn_mask])
0.96594778660612934
```

It's not completely surprising that we did well. We did fairly well in the dual class case, and since one will guess that the `talk.politics.guns` category is fairly orthogonal to the other two, we should probably do pretty well.

Label propagation with semi-supervised learning

Label propagation is a **semi-supervised** technique that makes use of the labeled and unlabeled data to learn about the unlabeled data. Quite often, data that will benefit from a classification algorithm is difficult to label. For example, labeling data might be very expensive, so only a subset is cost-effective to manually label. This said, there does seem to be slow, but growing, support for companies to hire taxonomists.

Getting ready

Another problem area is censored data. You can imagine a case where the frontier of time will affect your ability to gather labeled data. Say, for instance, you took measurements of patients and gave them an experimental drug. In some cases, you are able to measure the outcome of the drug, if it happens fast enough, but you might want to predict the outcome of the drugs that have a slower reaction time. The drug might cause a fatal reaction for some patients, and life-saving measures might need to be taken.

How to do it...

In order to represent the semi-supervised or censored data, we'll need to do a little data preprocessing. First, we'll walk through a simple example, and then we'll move on to some more difficult cases:

```
>>> from sklearn import datasets  
>>> d = datasets.load_iris()
```

Due to the fact that we'll be messing with the data, let's make copies and add an unlabeled member to the target name's copy. It'll make it easier to identify data later:

```
>>> x = d.data.copy()  
>>> y = d.target.copy()  
>>> names = d.target_names.copy()  
  
>>> names = np.append(names, ['unlabeled'])  
>>> names  
array(['setosa', 'versicolor', 'virginica',  
'unlabeled'],  
      dtype='|S10')
```

Now, let's update `y` with `-1`. This is the marker for the unlabeled case. This is also why we added `unlabeled` to the end of `names`:

```
>>> y[np.random.choice([True, False], len(y))] =  
-1
```

Our data now has a bunch of negative ones (-1) interspersed with the actual data:

```
>>> y[:10]
array([-1, -1, -1, -1,  0,  0, -1, -1,  0, -1])

>>> names[y[:10]]
array(['unlabeled', 'unlabeled', 'unlabeled',
'unlabeled', 'setosa',
'setosa', 'unlabeled', 'unlabeled',
'setosa', 'unlabeled'],
      dtype='|S10')
```

We clearly have a lot of unlabeled data, and the goal now is to use LabelPropagation to predict the labels:

```
>>> from sklearn import semi_supervised
>>> lp = semi_supervised.LabelPropagation()

>>> lp.fit(X, y)

LabelPropagation(alpha=1, gamma=20,
kernel='rbf', max_iter=30,
          n_neighbors=7, tol=0.001)
>>> preds = lp.predict(X)
>>> (preds == d.target).mean()
0.9866666666666669
```

Not too bad, though we did use all the data, so it's kind of cheating. Also, the `iris` dataset is a fairly separated dataset.

While we're at it, let's look at `LabelSpreading`, the "sister" class of `LabelPropagation`. We'll make the technical distinction between `LabelPropagation` and `LabelSpreading` in the *How it works...* section of this recipe, but it's easy to say that they are extremely similar:

```
>>> ls = semi_supervised.LabelSpreading()
```

`LabelSpreading` is more robust and noisy as observed from the way it works:

```
>>> ls.fit(X, y)
LabelSpreading(alpha=0.2, gamma=20,
kernel='rbf', max_iter=30,
n_neighbors=7, tol=0.001)

>>> (ls.predict(X) == d.target).mean()
0.9666666666666667
```

Don't consider the fact that the label-spreading algorithm missed one more as an indication and that it performs worse in general. The whole point is that we might give some ability to predict well on the training set and to work on a wider range of situations.

How it works...

Label propagation works by creating a graph of the data points, with weights placed on the edge equal to the following:

$$w_{ij}(\theta) = \frac{d_{ij}}{\theta^2}$$

The algorithm then works by labeled data points propagating their labels to the unlabeled data. This propagation is in part determined by edge weight.

The edge weights can be placed in a matrix of transition probabilities. We can iteratively determine a good estimate of the actual labels.

Chapter 5. Postmodel Workflow

This chapter will cover the following recipes:

- K-fold cross validation
- Automatic cross validation
- Cross validation with ShuffleSplit
- Stratified k-fold
- Poor man's grid search
- Brute force grid search
- Using dummy estimators to compare results
- Regression model evaluation
- Feature selection
- Feature selection on L1 norms
- Persisting models with joblib

Introduction

Even though by design the chapters are unordered, you could argue by virtue of the art of data science, we've saved the best for last.

For the most part, each recipe within this chapter is applicable to the various models we've worked with. In some ways, you can think about this chapter as tuning the parameters and features. Ultimately, we need to choose some criteria to determine the "best" model. We'll use various measures to define best. This is covered in the *Regression model evaluation* recipe. Then in the *Cross validation with ShuffleSplit* recipe, we will randomize the evaluation across subsets of the data to help avoid overfitting.

K-fold cross validation

In this recipe, we'll create, quite possibly, the most important post-model validation exercise—cross validation. We'll talk about k-fold cross validation in this recipe. There are several varieties of cross validation, each with slightly different randomization schemes. K-fold is perhaps one of the most well-known randomization schemes.

Getting ready

We'll create some data and then fit a classifier on the different folds. It's probably worth mentioning that if you can keep a holdout set, then that would be best. For example, we have a dataset where $N = 1000$. If we hold out 200 data points, then use cross validation between the other 800 points to determine the best parameters.

How to do it...

First, we'll create some fake data, then we'll examine the parameters, and finally, we'll look at the size of the resulting dataset:

```
>>> N = 1000
>>> holdout = 200

>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(1000, shuffle=True)
```

Now that we have the data, let's hold out 200 points, and then go through the fold scheme like we normally would:

```
>>> x_h, y_h = X[:holdout], y[:holdout]
>>> x_t, y_t = X[holdout:], y[holdout:]

>>> from sklearn.cross_validation import KFold
```

K-fold gives us the option of choosing how many folds we want, if we want the values to be indices or Booleans, if want to shuffle the dataset, and finally, the random state (this is mainly for reproducibility). Indices will actually be removed in later versions. It's assumed to be `True`.

Let's create the cross validation object:

```
>>> kfold = KFold(len(y_t), n_folds=4)
```

Now, we can iterate through the k-fold object:

```
>>> output_string = "Fold: {}, N_train: {},  
N_test: {}"  
  
>>> for i, (train, test) in enumerate(kfold):  
    print output_string.format(i,  
len(y_t[train]), len(y_t[test]))  
  
Fold: 0, N_train: 600, N_test: 200  
Fold: 1, N_train: 600, N_test: 200  
Fold: 2, N_train: 600, N_test: 200  
Fold: 3, N_train: 600, N_test: 200
```

Each iteration should return the same split size.

How it works...

It's probably clear, but k-fold works by iterating through the folds and holds out $\frac{1}{n_folds} * N$, where N for us was `len(y_t)`.

From a Python perspective, the cross validation objects have an iterator that can be accessed by using the `in` operator. Often times, it's useful to write a wrapper around a cross validation object that will iterate a subset of the data. For example, we may have a dataset that has repeated measures for data points or we may have a dataset with patients and each patient having measures.

We're going to mix it up and use pandas for this part:

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

>>> patients = np.repeat(np.arange(0, 100,
dtype=np.int8), 8)

>>> measurements = pd.DataFrame({'patient_id':
patients,
                                'ys': np.random.normal(0, 1,
800)})
```

Now that we have the data, we only want to hold out certain customers instead of data points:

```
>>> custids = np.unique(measurements.patient_id)
>>> customer_kfold = KFold(custids.size,
n_folds=4)

>>> output_string = "Fold: {}, N_train: {}, N_test: {}"

>>> for i, (train, test) in
enumerate(customer_kfold):
    train_cust_ids = custids[train]
    training =
measurements[measurements.patient_id.isin(
                train_cust_ids)]
    testing =
measurements[~measurements.patient_id.isin(
                train_cust_ids)]
    print output_string.format(i,
len(training), len(testing))
```

```
Fold: 0, N_train: 600, N_test: 200
Fold: 1, N_train: 600, N_test: 200
Fold: 2, N_train: 600, N_test: 200
Fold: 3, N_train: 600, N_test: 200
```

Automatic cross validation

We've looked at the using cross validation iterators that scikit-learn comes with, but we can also use a helper function to perform cross validation for use automatically. This is similar to how other objects in scikit-learn are wrapped by helper functions, pipeline for instance.

Getting ready

First, we'll need to create a sample classifier; this can really be anything, a decision tree, a random forest, whatever. For us, it'll be a random forest. We'll then create a dataset and use the cross validation functions.

How to do it...

First import the `ensemble` module and we'll get started:

```
>>> from sklearn import ensemble  
>>> rf =  
ensemble.RandomForestRegressor(max_features='auto')
```

Okay, so now, let's create some regression data:

```
>>> from sklearn import datasets  
>>> X, y = datasets.make_regression(10000, 10)
```

Now that we have the data, we can import the `cross_validation` module and get access to the functions we'll use:

```
>>> from sklearn import cross_validation  
  
>>> scores =  
cross_validation.cross_val_score(rf, X, y)  
  
>>> print scores  
[ 0.86823874  0.86763225  0.86986129]
```

How it works...

For the most part, this will delegate to the cross validation objects. One nice thing is that, the function will handle performing the cross validation in parallel.

We can activate verbose mode play by play:

```
>>> scores =
cross_validation.cross_val_score(rf, X, y,
verbose=3, cv=4)

[CV] no parameters to be set
[CV] no parameters to be set, score=0.872866 -
0.7s
[CV] no parameters to be set
[CV] no parameters to be set, score=0.873679 -
0.6s
[CV] no parameters to be set
[CV] no parameters to be set, score=0.878018 -
0.7s
[CV] no parameters to be set
[CV] no parameters to be set, score=0.871598 -
0.6s

[Parallel(n_jobs=1)]: Done    1 jobs          |
elapsed:    0.7s
[Parallel(n_jobs=1)]: Done    4 out of  4 |
elapsed:    2.6s finished
```

As we can see, during each iteration, we scored the

function. We also get an idea of how long the model runs.

It's also worth knowing that we can score our function predicated on which kind of model we're trying to fit. In other recipes, we've discussed how to create your own scoring function.

Cross validation with ShuffleSplit

ShuffleSplit is one of the simplest cross validation techniques. This cross validation technique will simply take a sample of the data for the number of iterations specified.

Getting ready

ShuffleSplit is another cross validation technique that is very simple. We'll specify the total elements in the dataset, and it will take care of the rest. We'll walk through an example of estimating the mean of a univariate dataset. This is somewhat similar to resampling, but it'll illustrate one reason why we want to use cross validation while showing cross validation.

How to do it...

First, we need to create the dataset. We'll use NumPy to create a dataset, where we know the underlying mean. We'll sample half of the dataset to estimate the mean and see how close it is to the underlying mean:

```
>>> import numpy as np

>>> true_loc = 1000
>>> true_scale = 10
>>> N = 1000

>>> dataset = np.random.normal(true_loc,
true_scale, N)

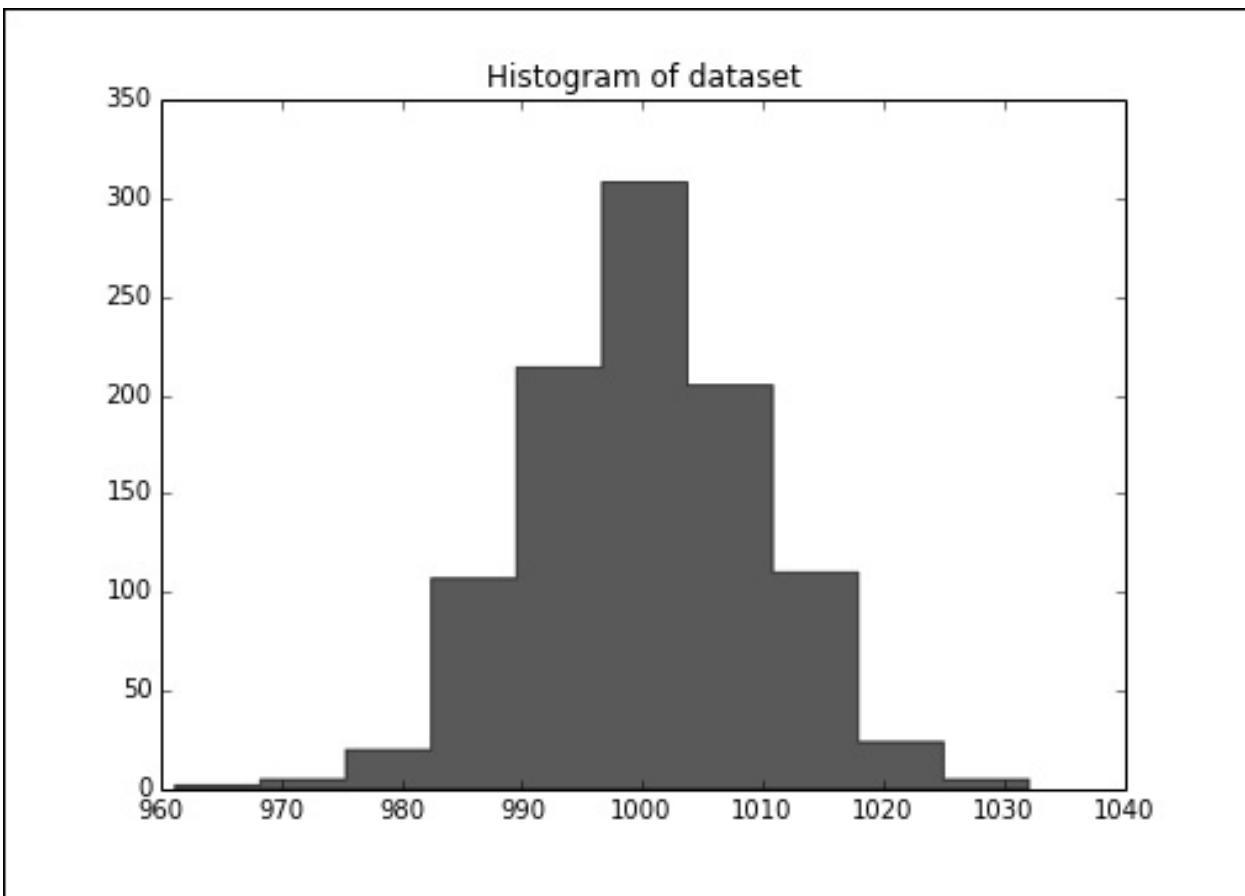
>>> import matplotlib.pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.hist(dataset, color='k', alpha=.65,
histtype='stepfilled');
>>> ax.set_title("Histogram of dataset");

>>> f.savefig("978-1-78398-948-5_06_06.png")
```

NumPy will give the following output:



Now, let's take the first half of the data and guess the mean:

```
>>> from sklearn import cross_validation  
  
>>> holdout_set = dataset[:500]  
>>> fitting_set = dataset[500:]  
  
>>> estimate = fitting_set[:N/2].mean()
```

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

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.set_title("True Mean vs Regular
Estimate")

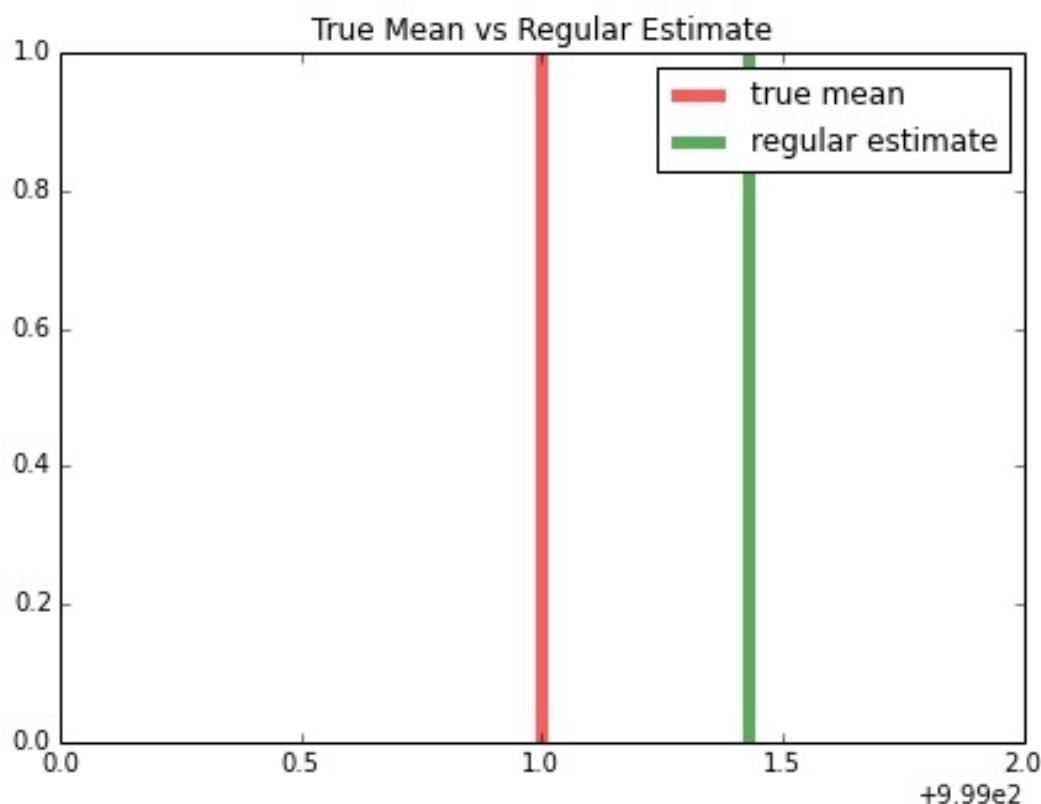
>>> ax.vlines(true_loc, 0, 1, color='r',
linestyles='--', lw=5,
alpha=.65, label='true mean')
>>> ax.vlines(estimate, 0, 1, color='g',
linestyles='--', lw=5,
alpha=.65, label='regular
estimate')

>>> ax.set_xlim(999, 1001)

>>> ax.legend()

>>> f.savefig("978-1-78398-948-5_06_07.png")
```

We'll get the following output:



Now, we can use ShuffleSplit to fit the estimator on several smaller datasets:

```
>>> from sklearn.cross_validation import  
ShuffleSplit
```

```
>>> shuffle_split =  
ShuffleSplit(len(fitting_set))
```

```
>>> mean_p = []

>>> for train, _ in shuffle_split:
    mean_p.append(fitting_set[train].mean())
    shuf_estimate = np.mean(mean_p)

>>> import matplotlib.pyplot as plt

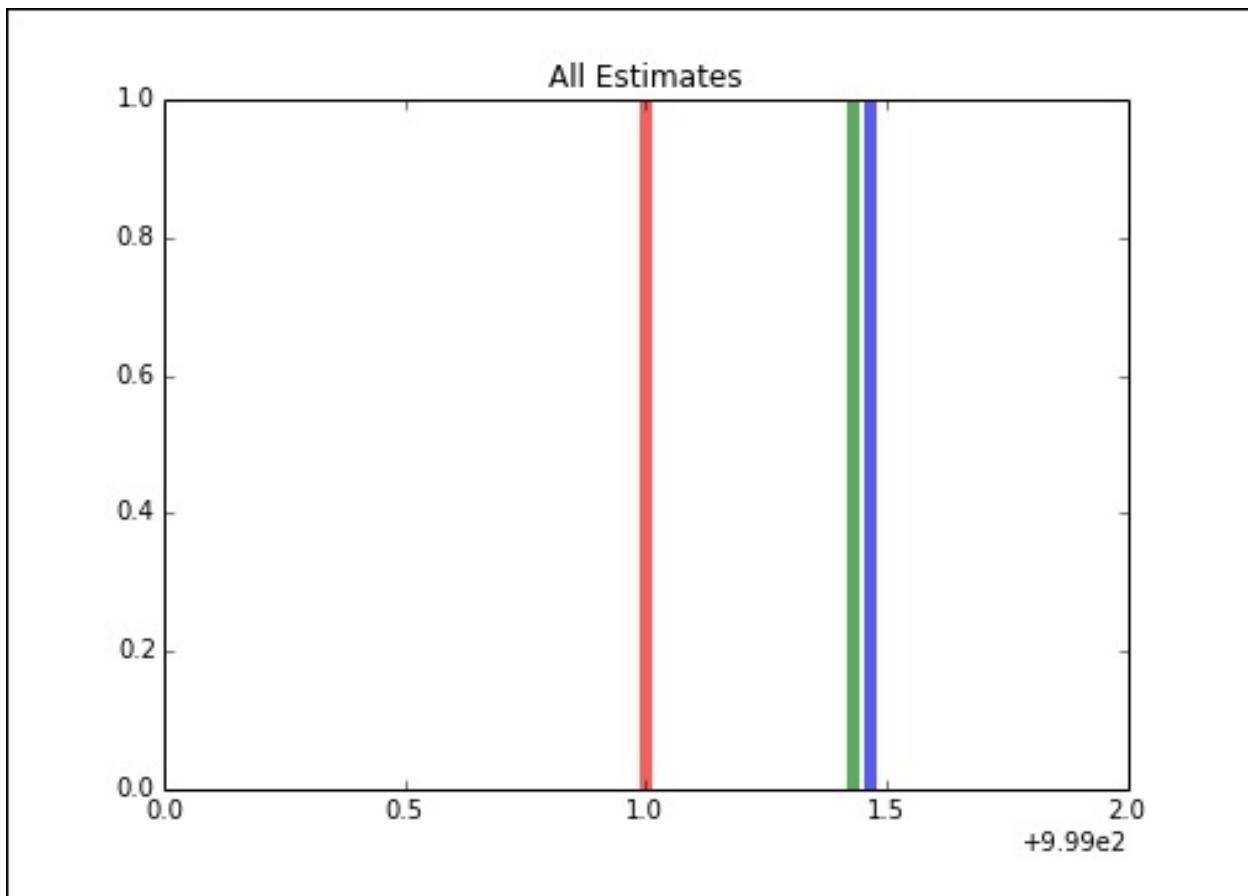
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.vlines(true_loc, 0, 1, color='r',
linestyles='--', lw=5,
alpha=.65, label='true mean')
>>> ax.vlines(estimate, 0, 1, color='g',
linestyles='--', lw=5,
alpha=.65, label='regular
estimate')
>>> ax.vlines(shuf_estimate, 0, 1, color='b',
linestyles='--', lw=5,
alpha=.65, label='shufflesplit
estimate')

>>> ax.set_title("All Estimates")
>>> ax.set_xlim(999, 1001)

>>> ax.legend(loc=3)
```

The output will be as follows:



As we can see, we got an estimate that was similar to what we expected, but we were able to take many samples to get that estimate.

Stratified k-fold

In this recipe, we'll quickly look at stratified k-fold valuation. We've walked through different recipes where the class representation was unbalanced in some manner. Stratified k-fold is nice because its scheme is specifically designed to maintain the class proportions.

Getting ready

We're going to create a small dataset. In this dataset, we will then use stratified k-fold validation. We want it small so that we can see the variation. For larger samples, it probably won't be as big of a deal.

We'll then plot the class proportions at each step to illustrate how the class proportions are maintained:

```
>>> from sklearn import datasets  
>>> X, y =  
datasets.make_classification(n_samples=int(1e3),  
weights=[1./11])
```

Let's check the overall class weight distribution:

```
>>> y.mean()  
0.9030000000000002
```

Roughly, 90.5 percent of the samples are 1, with the balance 0.

How to do it...

Let's create a stratified k-fold object and iterate it through each fold. We'll measure the proportion of `verse` that are 1. After that we'll plot the proportion of classes by the split number to see how and if it changes. This code will hopefully illustrate how this is beneficial. We'll also plot this code against a basic `ShuffleSplit`:

```
>>> from sklearn import cross_validation  
  
>>> n_folds = 50  
  
>>> strat_kfold =  
cross_validation.StratifiedKFold(y,  
n_folds=n_folds)  
>>> shuff_split =  
cross_validation.ShuffleSplit(n=len(y),  
n_iter=n_folds)  
  
>>> kfold_y_props = []  
>>> shuff_y_props = []  
  
>>> for (k_train, k_test), (s_train, s_test) in  
zip(strat_kfold, >>> shuff_split):  
    kfold_y_props.append(y[k_train].mean())  
    shuff_y_props.append(y[s_train].mean())
```

Now, let's plot the proportions over each fold:

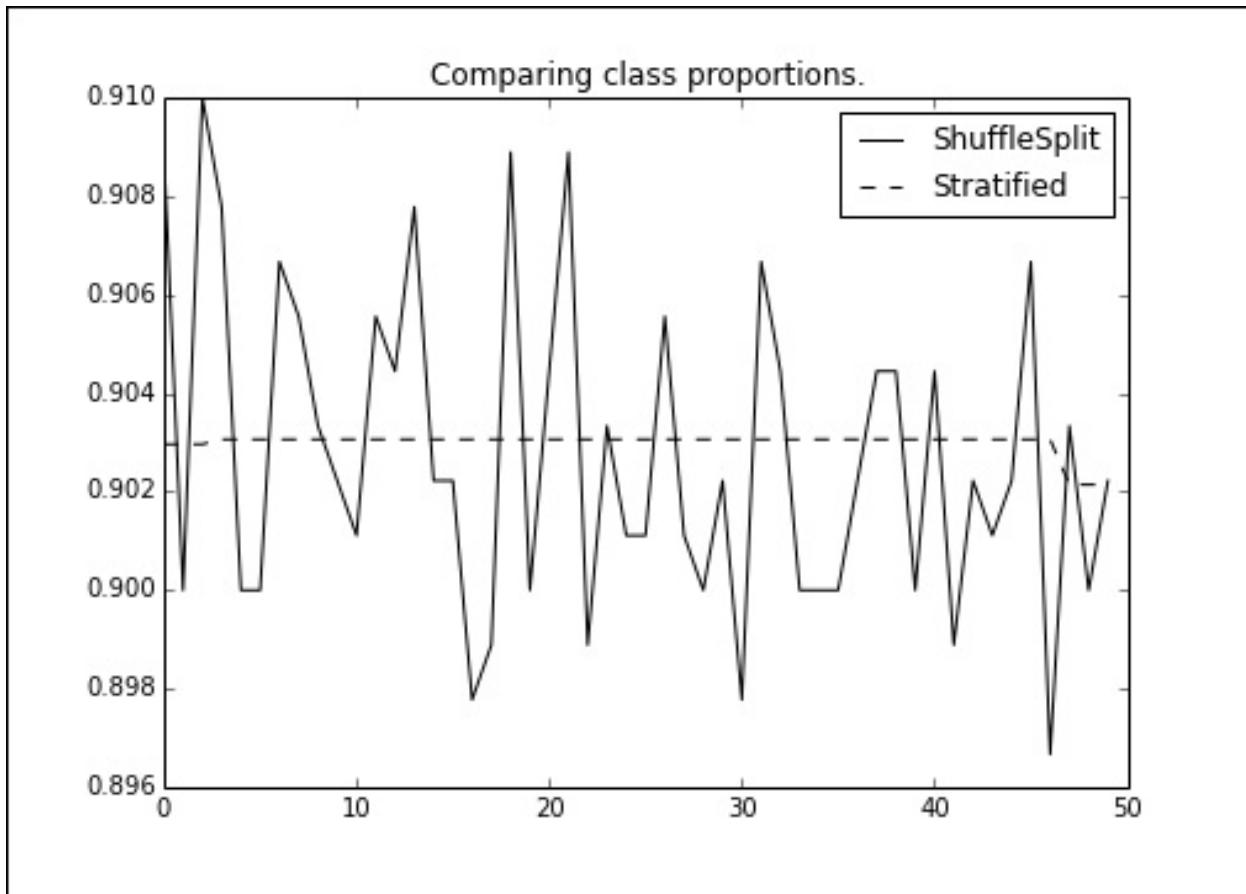
```
>>> import matplotlib.pyplot as plt
```

```
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.plot(range(n_folds), shuff_y_props,
label="ShuffleSplit",
color='k')
>>> ax.plot(range(n_folds), kfold_y_props,
label="Stratified",
color='k', ls='--')
>>> ax.set_title("Comparing class proportions.")

>>> ax.legend(loc='best')
```

The output will be as follows:



We can see that the proportion of each fold for stratified k-fold is stable across folds.

How it works...

Stratified k-fold works by taking the y value. First, getting the overall proportion of the classes, then intelligently splitting the training and test set into the proportions. This will generalize to multiple labels:

```
>>> import numpy as np

>>> three_classes = np.random.choice([1,2,3], p=[.1, .4, .5],
                                     size=1000)

>>> import itertools as it

>>> for train, test in
cross_validation.StratifiedKFold(three_classes,
5):
    print np.bincount(three_classes[train])

[ 0  90 314 395]
[ 0  90 314 395]
[ 0  90 314 395]
[ 0  91 315 395]
[ 0  91 315 396]
```

As we can see, we got roughly the sample sizes of each class for our training and testing proportions.

Poor man's grid search

In this recipe, we're going to introduce grid search with basic Python, though we will use sklearn for the models and matplotlib for the visualization.

Getting ready

In this recipe, we will perform the following tasks:

- Design a basic search grid in the parameter space
- Iterate through the grid and check the loss/score function at each point in the parameter space for the dataset
- Choose the point in the parameter space that minimizes/maximizes the evaluation function

Also, the model we'll fit is a basic decision tree classifier. Our parameter space will be 2 dimensional to help us with the visualization:

```
criteria = {gini, entropy}
```

```
max_features = {auto, log2, None}
```

The parameter space will then be the Cartesian product of the those two sets:

```
parameter space = criteria × max_features
```

We'll see in a bit how we can iterate through this space with `itertools`.

Let's create the dataset and then get started:

```
>>> from sklearn import datasets  
>>> X, y =  
datasets.make_classification(n_samples=2000,  
n_features=10)
```

How to do it...

Earlier we said that we'd use grid search to tune two parameters—`criterion` and `max_features`. We need to represent those as Python sets, and then use `itertools` product to iterate through them:

```
>>> criteria = {'gini', 'entropy'}
>>> max_features = {'auto', 'log2', None}
>>> import itertools as it
>>> parameter_space = it.product(criteria,
max_features)
```

Great! So now that we have the parameter space, let's iterate through it and check the accuracy of each model as specified by the parameters. Then, we'll store that accuracy so that we can compare different parameter spaces. We'll also use a test and train split of 50, 50:

```
import numpy as np
train_set = np.random.choice([True, False],
size=len(y))
from sklearn.tree import DecisionTreeClassifier
accuracies = {}
for criterion, max_feature in parameter_space:
    dt =
    DecisionTreeClassifier(criterion=criterion,
                           max_features=max_feature)
    dt.fit(X[train_set], y[train_set])
    accuracies[(criterion, max_feature)] =
```

```

(dt.predict(X[~train_set])
 ==

y[~train_set]).mean()
>>> accuracies
{('entropy', None): 0.974609375, ('entropy',
'auto'): 0.9736328125, ('entropy', 'log2'):
0.962890625, ('gini', None): 0.9677734375,
('gini', 'auto'): 0.9638671875, ('gini',
'log2'): 0.96875}

```

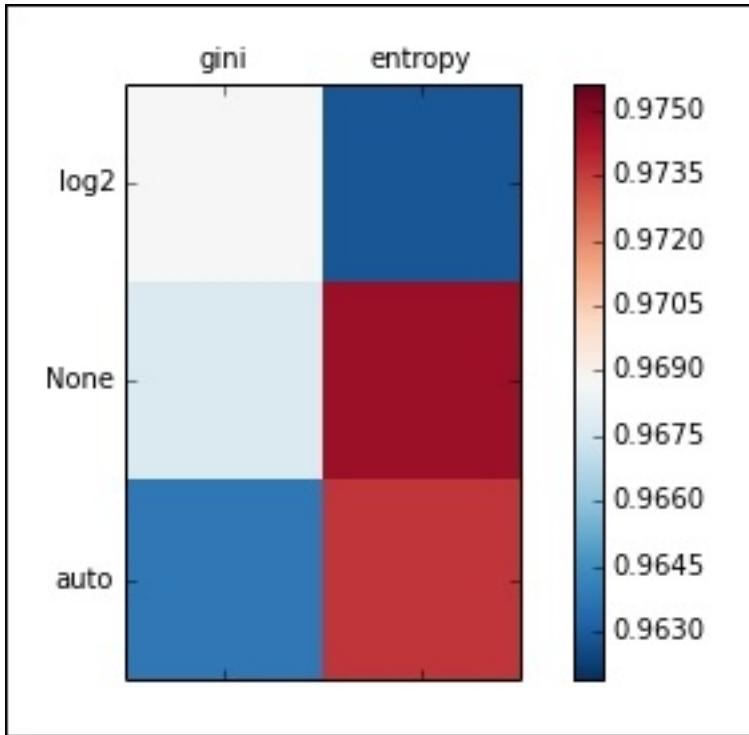
So we now have the accuracies and its performance. Let's visualize the performance:

```

>>> from matplotlib import pyplot as plt
>>> from matplotlib import cm
>>> cmap = cm.RdBu_r
>>> f, ax = plt.subplots(figsize=(7, 4))
>>> ax.set_xticklabels([''] + list(criteria))
>>> ax.set_yticklabels([''] +
list(max_features))
>>> plot_array = []
>>> for max_feature in max_features:
    m = []
>>> for criterion in criteria:
    m.append(accuracies[(criterion,
max_feature)])
    plot_array.append(m)
>>> colors = ax.matshow(plot_array,
vmin=np.min(accuracies.values()) -
0.001,
vmax=np.max(accuracies.values()) + 0.001,
cmap=cmap)
>>> f.colorbar(colors)

```

The following is the output:



It's fairly easy to see which one performed best here. Hopefully, you can see how this process can be taken to the further stage with a brute force method.

How it works...

This works fairly simply, we just have to perform the following steps:

1. Choose a set of parameters.
2. Iterate through them and find the accuracy of each step.
3. Find the best performer by visual inspection.

Brute force grid search

In this recipe, we'll do an exhaustive grid search through scikit-learn. This is basically the same thing we did in the previous recipe, but we'll utilize built-in methods.

We'll also walk through an example of performing randomized optimization. This is an alternative to brute force search. Essentially, we're trading computer cycles to make sure that we search the entire space. We were fairly calm in the last recipe. However, you could imagine a model that has several steps, first imputation for fix missing data, then PCA reduce the dimensionality to classification. Your parameter space could get very large, very fast; therefore, it can be advantageous to only search a part of that space.

Getting ready

To get started, we'll need to perform the following steps:

1. Create some classification data.
2. We'll then create a `LogisticRegression` object that will be the model we're fitting.
3. After that, we'll create the search objects, `GridSearch` and `RandomizedSearchCV`.

How to do it...

Run the following code to create some classification data:

```
>>> from sklearn.datasets import  
make_classification  
  
>>> X, y = make_classification(1000,  
n_features=5)
```

Now, we'll create our logistic regression object:

```
>>> from sklearn.linear_model import  
LogisticRegression  
  
>>> lr = LogisticRegression(class_weight='auto')
```

We need to specify the parameters we want to search. For GridSearch, we can just specify the ranges that we care about, but for RandomizedSearchCV, we'll need to actually specify the distribution over the same space from which to sample:

```
>>> lr.fit(X, y)  
  
LogisticRegression(C=1.0, class_weight={0: 0.25,  
1: 0.75}, dual=False,  
                    fit_intercept=True,  
intercept_scaling=1,  
                    penalty='l2',  
random_state=None, tol=0.0001)
```

```
>>> grid_search_params = {'penalty': ['l1',  
'l2'],  
                           'C': [1, 2, 3, 4]}
```

The only change we'll need to make is to describe the C parameter as a probability distribution. We'll keep it simple right now, though we will use `scipy` to describe the distribution:

```
>>> import scipy.stats as st  
>>> import numpy as np  
  
>>> random_search_params = {'penalty': ['l1',  
'l2'],  
                           'C': st.randint(1,  
4) }
```

How it works...

Now, we'll fit the classifier. This works by passing `lr` to the parameter search objects:

```
>>> from sklearn.grid_search import  
GridSearchCV, RandomizedSearchCV
```

```
>>> gs = GridSearchCV(lr, grid_search_params)
```

`GridSearchCV` implements the same API as the other models:

```
>>> gs.fit(X, y)
```

```
GridSearchCV(cv=None,  
estimator=LogisticRegression(C=1.0,  
                             class_weight='auto', dual=False,  
                             fit_intercept=True,  
                             intercept_scaling=1, penalty='l2',  
                             random_state=None,  
                             tol=0.0001), fit_params={},  
iid=True, loss_func=None,  
n_jobs=1, param_grid={'penalty':  
['l1', 'l2'],  
'C': [1, 2, 3, 4]},  
pre_dispatch='2*n_jobs', refit=True,  
score_func=None, scoring=None,  
verbose=0)
```

As we can see with the `param_grid` parameter, our `penalty` and `C` are both arrays.

To access the scores, we can use the `grid_scores_` attribute of the grid search. We also want to find the optimal set of parameters. We can also look at the marginal performance of the grid search:

```
>>> gs.grid_scores_
[mean: 0.90300, std: 0.01192, params:
 {'penalty': 'l1', 'C': 1},
 mean: 0.90100, std: 0.01258, params:
 {'penalty': 'l2', 'C': 1},
 mean: 0.90200, std: 0.01117, params:
 {'penalty': 'l1', 'C': 2},
 mean: 0.90100, std: 0.01258, params:
 {'penalty': 'l2', 'C': 2},
 mean: 0.90200, std: 0.01117, params:
 {'penalty': 'l1', 'C': 3},
 mean: 0.90100, std: 0.01258, params:
 {'penalty': 'l2', 'C': 3},
 mean: 0.90100, std: 0.01258, params:
 {'penalty': 'l1', 'C': 4},
 mean: 0.90100, std: 0.01258, params:
 {'penalty': 'l2', 'C': 4}]
```

We might want to get the max score:

```
>>> gs.grid_scores_[1][1]
```

```
0.9010000000000002
```

```
>>> max(gs.grid_scores_, key=lambda x: x[1])  
  
mean: 0.90300, std: 0.01192, params: {'penalty':  
'l1', 'C': 1}
```

The parameters obtained are the best choices for our logistic regression.

Using dummy estimators to compare results

This recipe is about creating fake estimators; this isn't the pretty or exciting stuff, but it is worthwhile to have a reference point for the model you'll eventually build.

Getting ready

In this recipe, we'll perform the following tasks:

1. Create some data random data.
2. Fit the various dummy estimators.

We'll perform these two steps for regression data and classification data.

How to do it...

First, we'll create the random data:

```
>>> from sklearn.datasets import  
make_regression, make_classification  
# classification if for later  
  
>>> X, y = make_regression()  
  
>>> from sklearn import dummy  
  
>>> dumdum = dummy.DummyRegressor()  
  
>>> dumdum.fit(X, y)  
  
DummyRegressor(constant=None, strategy='mean')
```

By default, the estimator will predict by just taking the mean of the values and predicting the mean values:

```
>>> dumdum.predict(X) [:5]  
  
array([ 2.23297907,  2.23297907,  2.23297907,  
       2.23297907,  
       2.23297907])
```

There are other two other strategies we can try. We can predict a supplied constant (refer to `constant=None` from the preceding command). We can also predict the median value.

Supplying a constant will only be considered if strategy is "constant".

Let's have a look:

```
>>> predictors = [ ("mean", None) ,
                    ("median", None) ,
                    ("constant", 10) ]

>>> for strategy, constant in predictors:
        dumdum =
    dummy.DummyRegressor(strategy=strategy,
                          constant=constant)
>>> dumdum.fit(X, y)

>>> print "strategy: {}".format(strategy),
", ".join(map(str,
              dumdum.predict(X)[:5]))


strategy: mean
2.23297906733, 2.23297906733, 2.23297906733, 2.23297906733, 2.23297906733
strategy: median
20.38535248, 20.38535248, 20.38535248, 20.38535248, 20.38535248
strategy: constant
10.0, 10.0, 10.0, 10.0, 10.0
```

We actually have four options for classifiers. These strategies are similar to the continuous case, it's just slanted toward classification problems:

```
>>> predictors = [ ("constant", 0) ,
```

```
("stratified", None),  
("uniform", None),  
("most_frequent", None)]
```

We'll also need to create some classification data:

```
>>> X, y = make_classification()  
  
>>> for strategy, constant in predictors:  
    dumdum =  
dummy.DummyClassifier(strategy=strategy,  
                      constant=constant)  
    dumdum.fit(X, y)  
    print "strategy: {}".format(strategy),  
", ".join(map(str,  
            dumdum.predict(X)[:5]))  
  
strategy: constant 0,0,0,0,0  
strategy: stratified 1,0,0,1,0  
strategy: uniform 0,0,0,1,1  
strategy: most_frequent 1,1,1,1,1
```

How it works...

It's always good to test your models against the simplest models and that's exactly what the dummy estimators give you. For example, imagine a fraud model. In this model, only 5 percent of the data set is fraud. Therefore, we can probably fit a pretty good model just by never guessing any fraud.

We can create this model by using the stratified strategy, using the following command. We can also get a good example of why class imbalance causes problems:

```
>>> X, y = make_classification(20000, weights=[.95, .05])  
  
>>> dumdum =  
dummy.DummyClassifier(strategy='most_frequent')  
  
>>> dumdum.fit(X, y)  
DummyClassifier(constant=None,  
random_state=None, strategy='most_frequent')  
  
>>> from sklearn.metrics import accuracy_score  
  
>>> print accuracy_score(y, dumdum.predict(X))  
0.94575
```

We were actually correct very often, but that's not the

point. The point is that this is our baseline. If we cannot create a model for fraud that is more accurate than this, then it isn't worth our time.

Regression model evaluation

We learned about quantifying the error in classification, now we'll discuss quantifying the error for continuous problems. For example, we're trying to predict an age, not a gender.

Getting ready

Like the classification, we'll fake some data, then plot the change. We'll start simple, then build up the complexity. The data will be a simulated linear model:

```
m = 2  
b = 1  
  
y = lambda x: m*x+b
```

Also, let's get our modules loaded:

```
>>> import numpy as np  
>>> import matplotlib.pyplot as plt  
>>> from sklearn import metrics
```

How to do it...

We will be performing the following actions:

1. Use 'y' to generate 'y_actual'.
2. Use 'y_actual' plus some `err` to generate 'y_prediction'.
3. Plot the differences.
4. Walk through various metrics and plot some of them.

Let's take care of steps 1 and 2 at the same time and just have a function do the work for us. This will be almost the same thing we just saw, but we'll add the ability to specify an error (or bias if a constant):

```
>>> def data(x, m=2, b=1, e=None, s=10):
        """
        Args:
            x: The x value
            m: Slope
            b: Intercept
            e: Error, optional, True will give
random error
        """

        if e is None:
            e_i = 0
        elif e is True:
            e_i = np.random.normal(0, s, len(xs))
        else:
```

```
e_i = e

return x * m + b + e_i
```

Now that we have the function, let's define `y_hat` and `y_actual`. We'll do it in a convenient way:

```
>>> from functools import partial

>>> N = 100
>>> xs = np.sort(np.random.rand(N)*100)

>>> y_pred_gen = partial(data, x=xs, e=True)
>>> y_true_gen = partial(data, x=xs)

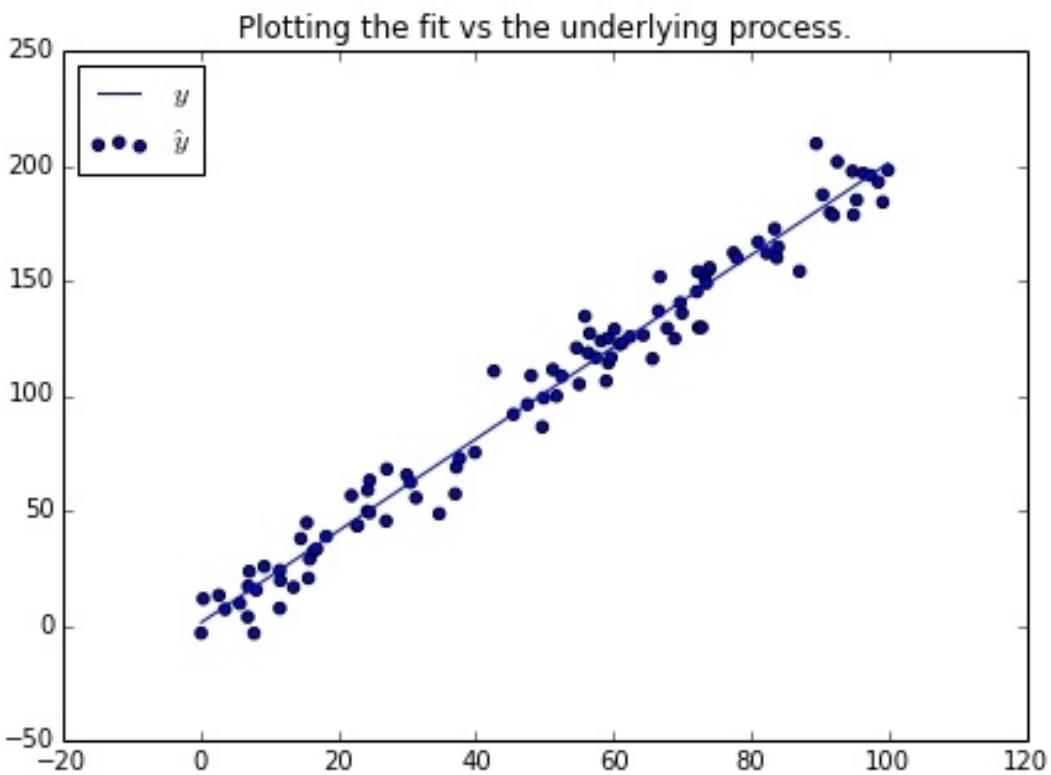
>>> y_pred = y_pred_gen()
>>> y_true = y_true_gen()

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.set_title("Plotting the fit vs the
underlying process.")
>>> ax.scatter(xs, y_pred, label=r'$\hat{y}$')
>>> ax.plot(xs, y_true, label=r'$y$')

>>> ax.legend(loc='best')
```

The output for this code is as follows:



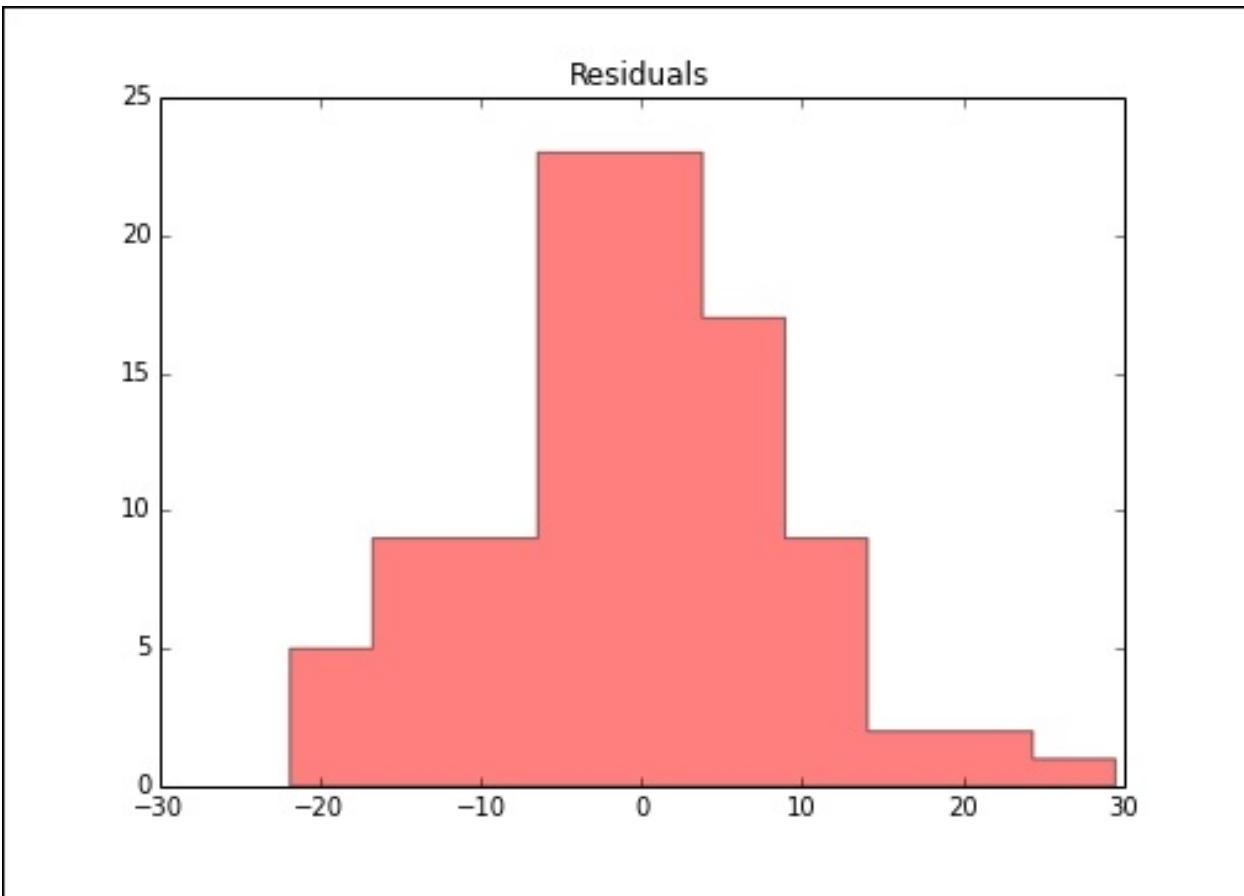
Just to confirm the output, we'd be working with the classical residuals:

```
>>> e_hat = y_pred - y_true

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.set_title("Residuals")
>>> ax.hist(e_hat, color='r', alpha=.5,
histtype='stepfilled')
```

The output for the residuals is as follows:



So that looks good now.

How it works...

Now let's move to the metrics.

First, a metric is the mean squared error:

$$MSE(y_{true}, y_{pred}) = E\left(\left(y_{true} - y_{pred}\right)^2\right)$$

You can use the following code to find the value of the mean squared error:

```
>>> metrics.mean_squared_error(y_true, y_pred)
```

```
93.342352628475368
```

You'll notice that this code will penalize large errors more than small errors. It's important to remember that all we're doing here is applying what probably was the cost function for the model on the test data.

Another option is the mean absolute deviation. We need to take the absolute value of the difference, if we don't, our value will probably be fairly close to zero, the mean of the distribution:

$$MAD(y_{true}, y_{pred}) = E(|y_{true} - y_{pred}|)$$

The final option is R^2 , this is 1 minus the ratio of squared errors for the overall mean and the fit model. As the ratio tends to 0, the R^2 tends to 1:

```
>>> metrics.r2_score(y_true, y_pred)
```

```
0.9729312117010761
```

R^2 is deceptive; it cannot give the clearest sense of the accuracy of the model.

Feature selection

This recipe along with the two following it will be centered around automatic feature selection. I like to think of this as the feature analogue of parameter tuning. In the same way that we cross-validate to find an appropriately general parameter, we can find an appropriately general subset of features. This will involve several different methods.

The simplest idea is univariate selection. The other methods involve working with a combination of features.

An added benefit to feature selection is that it can ease the burden on the data collection. Imagine that you have built a model on a very small subset of the data. If all goes well, you might want to scale up to predict the model on the entire subset of data. If this is the case, you can ease the engineering effort of data collection at that scale.

Getting ready

With univariate feature selection, scoring functions will come to the forefront again. This time, they will define the comparable measure by which we can eliminate features.

In this recipe, we'll fit a regression model with a few 10,000 features, but only 1,000 points. We'll walk through the various univariate feature selection methods:

```
>>> from sklearn import datasets  
>>> X, y = datasets.make_regression(1000, 10000)
```

Now that we have the data, we will compare the features that are included with the various methods. This is actually a very common situation when you're dealing in text analysis or some areas of bioinformatics.

How to do it...

First, we need to import the `feature_selection` module:

```
>>> from sklearn import feature_selection  
>>> f, p = feature_selection.f_regression(x, y)
```

Here, `f` is the f score associated with each linear model fit with just one of the features. We can then compare these features and based on this comparison, we can cull features. `p` is also the p value associated with that f value.

In statistics, the `p` value is the probability of a value more extreme than the current value of the test statistic. Here, the `f` value is the test statistic:

```
>>> f[:5]  
array([ 1.06271357e-03,  2.91136869e+00,  
       1.01886922e+00,  
       2.22483130e+00,  4.67624756e-01])  
>>> p[:5]  
array([ 0.97400066,  0.08826831,  0.31303204,  
       0.1361235,   0.49424067])
```

As we can see, many of the `p` values are quite large. We would rather want that the `p` values be quite small. So, we can grab NumPy out of our tool box and choose all the `p` values less than .05. These will be the features we'll use for the analysis:

```
>>> import numpy as np  
>>> idx = np.arange(0, X.shape[1])  
>>> features_to_keep = idx[p < .05]  
>>> len(features_to_keep)
```

501

As you can see, we're actually keeping a relatively large amount of features. Depending on the context of the model, we can tighten this p value. This will lessen the number of features kept.

Another option is using the `VarianceThreshold` object. We've learned a bit about it, but it's important to understand that our ability to fit models is largely based on the variance created by features. If there is no variance, then our features cannot describe the variation in the dependent variable. A nice feature of this, as per the documentation, is that because it does not use the outcome variable, it can be used for unsupervised cases.

We will need to set the threshold for which we eliminate features. In order to do that, we just take the median of the feature variances and supply that:

```
>>> var_threshold =  
feature_selection.VarianceThreshold(np.median(np  
.var(X, axis=1)))  
  
>>> var_threshold.fit_transform(X).shape
```

(1000, 4835)

As we can see, we eliminated roughly half the features, more or less what we would expect.

How it works...

In general, all these methods work by fitting a basic model with a single feature. Depending on whether we have a classification problem or a regression problem, we can use the appropriate scoring function.

Let's look at a smaller problem and visualize how feature selection will eliminate certain features. We'll use the same scoring function from the first example, but just 20 features:

```
>>> x, y = datasets.make_regression(10000, 20)

>>> f, p = feature_selection.f_regression(x, y)
```

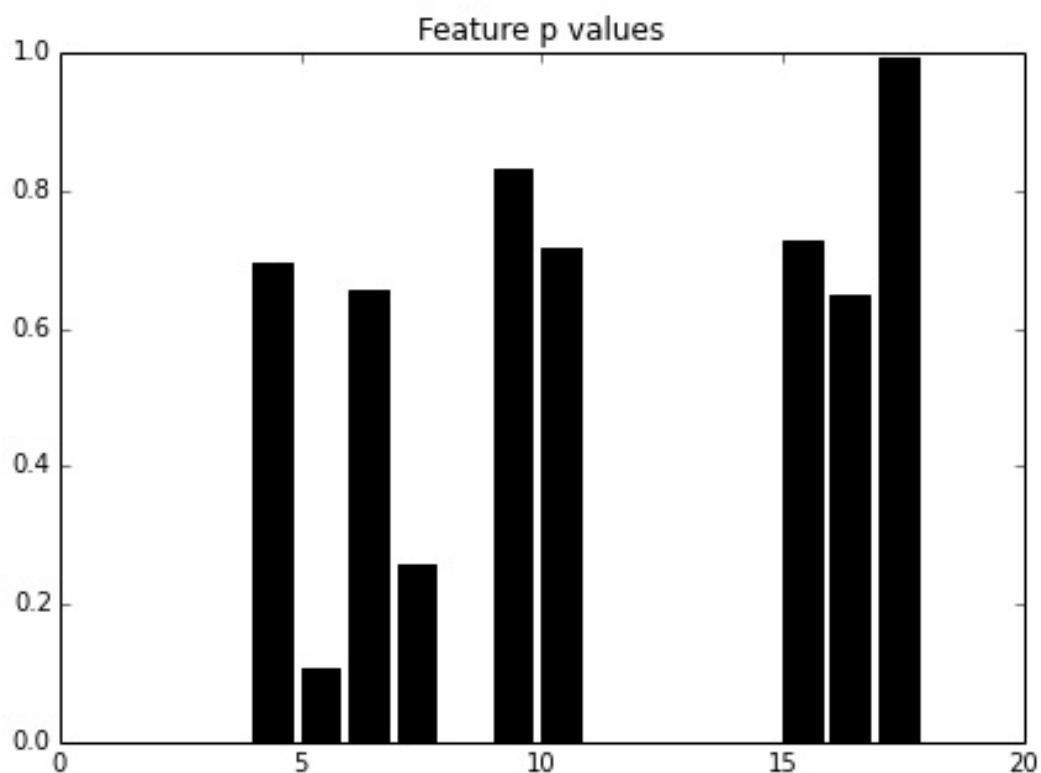
Now let's plot the p values of the features, we can see which feature will be eliminated and which will be kept:

```
>>> from matplotlib import pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.bar(np.arange(20), p, color='k')
>>> ax.set_title("Feature p values")
```

The output will be as follows:



As we can see, many of the features won't be kept, but several will be.

Feature selection on L1 norms

We're going to work with some ideas similar to those we saw in the recipe on Lasso Regression. In that recipe, we looked at the number of features that had zero coefficients.

Now we're going to take this a step further and use the sparseness associated with L1 norms to preprocess the features.

Getting ready

We'll use the diabetes dataset to fit a regression. First, we'll fit a basic `LinearRegression` model with a `ShuffleSplit` cross validation. After we do that, we'll use `LassoRegression` to find the coefficients that are 0 when using an `L1` penalty. This hopefully will help us to avoid overfitting, which means that the model is too specific to the data it was trained on. To put this another way, the model, if overfit, does not generalize well to outside data.

We're going to perform the following steps:

1. Load the dataset.
2. Fit a basic linear regression model.
3. Use feature selection to remove uninformative features.
4. Refit the linear regression and check to see how well it fits compared with the fully featured model.

How to do it...

First, let's get the dataset:

```
>>> import sklearn.datasets as ds  
>>> diabetes = ds.load_diabetes()
```

Let's create the `LinearRegression` object:

```
>>> from sklearn import linear_model  
>>> lr = linear_model.LinearRegression()
```

Let's also import the `metrics` module for the `mean_squared_error` function and the `cross_validation` module for the `ShuffleSplit` cross validation scheme:

```
>>> from sklearn import metrics  
>>> from sklearn import cross_validation
```

```
>>> shuff =  
cross_validation.ShuffleSplit(diabetes.target.size)
```

Now, let's fit the model, and we'll keep track of the mean squared error for each iteration of `ShuffleSplit`:

```
>>> mses = []  
>>> for train, test in shuff:  
    train_X = diabetes.data[train]  
    train_y = diabetes.target[train]
```

```
test_X = diabetes.data[~train]
test_y = diabetes.target[~train]

lr.fit(train_X, train_y)

mses.append(metrics.mean_squared_error(test_y,
                                         lr.predict(test_X)))

>>> np.mean(mses)

2856.366626198198
```

So now that we have the regular fit, let's check it after we eliminate any features with a zero for the coefficient. Let's fit the Lasso Regression:

```
>>> from sklearn import feature_selection
>>> from sklearn import cross_validation

>>> cv = linear_model.LassoCV()
>>> cv.fit(diabetes.data, diabetes.target)
>>> cv.coef_

array([-0. , -226.2375274 ,  526.85738059,
       314.44026013,
      -196.92164002,  1.48742026,
     -151.78054083, 106.52846989,
      530.58541123,  64.50588257])
```

We'll remove the first feature, I'll use a NumPy array to

represent the columns that are to be included in the model:

```
>>> import numpy as np
>>> columns = np.arange(diabetes.data.shape[1])
[cv.coef_ != 0]
>>> columns
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Okay, so now we'll fit the model with the specific features (see the columns in the following code block):

```
>>> l1mses = []

>>> for train, test in shuffle:
    train_X = diabetes.data[train][:, 
columns]
    train_y = diabetes.target[train]

    test_X = diabetes.data[~train][:, 
columns]
    test_y = diabetes.target[~train]

    lr.fit(train_X, train_y)

l1mses.append(metrics.mean_squared_error(test_y,
                                         lr.predict(test_X)))

>>> np.mean(l1mses)
2861.0763924492171
>>> np.mean(l1mses) - np.mean(mses)
```

4.7097662510191185

As we can see, even though we get an uninformative feature, the model still fits worse. This isn't always the case. In the next section, we'll compare a fit between models where there are many uninformative features.

How it works...

First, we're going to create a regression dataset with many uninformative features:

```
>>> X, y = ds.make_regression(noise=5)
```

Let's fit a normal regression:

```
>>> mses = []
```

```
>>> shuff =
cross_validation.ShuffleSplit(y.size)
```

```
>>> for train, test in shuff:
    train_X = X[train]
    train_y = y[train]

    test_X = X[~train]
    test_y = y[~train]

    lr.fit(train_X, train_y)
```

```
mses.append(metrics.mean_squared_error(test_y,
                                         lr.predict(test_X)))
```

```
>>> np.mean(mses)
```

```
879.75447864034209
```

Now, we can walk through the same process for Lasso

regression:

```
>>> cv.fit(X, y)

LassoCV(alphas=None, copy_X=True, cv=None,
eps=0.001,
    fit_intercept=True, max_iter=1000,
n_alphas=100,
    n_jobs=1, normalize=False,
positive=False, precompute='auto',
    tol=0.0001, verbose=False)
```

We'll create the columns again. This is a nice pattern that will allow us to specify the features we want to include:

```
>>> import numpy as np
>>> columns = np.arange(X.shape[1])[cv.coef_ != 0]
>>> columns[:5]
array([11, 15, 17, 20, 21,])

>>> mses = []
>>> shuff =
cross_validation.ShuffleSplit(y.size)

>>> for train, test in shuff:
    train_X = X[train][:, columns]
    train_y = y[train]

    test_X = X[~train][:, columns]
    test_y = y[~train]

    lr.fit(train_X, train_y)
```

```
mses.append(metrics.mean_squared_error(test_y,  
                                         lr.predict(test_X)))  
  
>>> np.mean(mses)  
  
15.755403220117708
```

As we can see, we get an extreme improvement in the fit of the model. This just exemplifies that we need to be cognizant that not all the models need to be or should be thrown into the model.

Persisting models with joblib

In this recipe, we're going to show how you can keep your model around for a later usage. For example, you might want to actually use a model to predict the outcome and automatically make a decision.

Getting ready

In this recipe, we will perform the following tasks:

1. Fit the model that we will persist.
2. Import joblib and save the model.

How to do it...

To persist models with joblib, the following code can be used:

```
>>> from sklearn import datasets, tree

>>> X, y = datasets.make_classification()
>>> dt = tree.DecisionTreeClassifier()
>>> dt.fit(X, y)

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')

>>> from sklearn.externals import joblib

>>> joblib.dump(dt, "dtree.clf")

['dtree.clf',
'dtree.clf_01.npy',
'dtree.clf_02.npy',
'dtree.clf_03.npy',
'dtree.clf_04.npy']
```

How it works...

The preceding code works by saving the state of the object that can be reloaded into a scikit-learn object. It's important to note that the state of model will have varying levels of complexity, given the model type.

For simplicity sake, consider that all we'd need to save is the way to predict the outcome for the given inputs. Well, for regression that would be easy, a little matrix algebra and we're done. However, for models like random forest, where we could have many trees, and those trees could be of various complexity levels, regression is difficult.

There's more...

We can check the size of decision tree versus random forest:

```
>>> from sklearn import ensemble

>>> rf = ensemble.RandomForestClassifier()
>>> rf.fit(X, y)

RandomForestClassifier(bootstrap=True,
compute_importances=None,
                           criterion='gini',
max_depth=None,
                           max_features='auto',
max_leaf_nodes=None,
                           min_density=None,
min_samples_leaf=1,
                           min_samples_split=2,
n_estimators=10,
                           n_jobs=1,
oob_score=False, random_state=None,
                           verbose=0)
```

I'm going to omit the output, but in total, there were 52 files outputted on my machine:

```
>>> joblib.dump(rf, "rf.clf")
['rf.clf',
 'rf.clf_01.npy',
 'rf.clf_02.npy',
 'rf.clf_03.npy',
```

```
'rf.clf_04.npy',  
'rf.clf_05.npy',  
'rf.clf_06.npy',...]
```

Part 3. Module 3

Mastering Machine Learning with scikit-learn

Apply effective learning algorithms to real-world problems using scikit-learn

Chapter 1. The Fundamentals of Machine Learning

In this chapter we will review the fundamental concepts in machine learning. We will discuss applications of machine learning algorithms, the supervised-unsupervised learning spectrum, uses of training and testing data, and model evaluation. Finally, we will introduce scikit-learn, and install the tools required in subsequent chapters.

Our imagination has long been captivated by visions of machines that can learn and imitate human intelligence. While visions of general artificial intelligence such as Arthur C. Clarke's HAL and Isaac Asimov's Sonny have yet to be realized, software programs that can acquire new knowledge and skills through experience are becoming increasingly common. We use such machine learning programs to discover new music that we enjoy, and to quickly find the exact shoes we want to purchase online. Machine learning programs allow us to dictate commands to our smartphones and allow our thermostats to set their own temperatures. Machine learning programs can decipher sloppily-written mailing addresses better than

humans, and guard credit cards from fraud more vigilantly. From investigating new medicines to estimating the page views for versions of a headline, machine learning software is becoming central to many industries. Machine learning has even encroached on activities that have long been considered uniquely human, such as writing the sports column recapping the Duke basketball team's loss to UNC.

Machine learning is the design and study of software artifacts that use past experience to make future decisions; it is the study of programs that learn from data. The fundamental goal of machine learning is to *generalize*, or to induce an unknown rule from examples of the rule's application. The canonical example of machine learning is spam filtering. By observing thousands of emails that have been previously labeled as either spam or ham, spam filters learn to classify new messages.

Arthur Samuel, a computer scientist who pioneered the study of artificial intelligence, said that machine learning is "the study that gives computers the ability to learn without being explicitly programmed." Throughout the 1950s and 1960s, Samuel developed programs that played checkers. While the rules of checkers are simple, complex strategies are required to defeat skilled opponents. Samuel never explicitly programmed these strategies, but through

the experience of playing thousands of games, the program learned complex behaviors that allowed it to beat many human opponents.

A popular quote from computer scientist Tom Mitchell defines machine learning more formally: "A program can be said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E ." For example, assume that you have a collection of pictures. Each picture depicts either a dog or cat. A task could be sorting the pictures into separate collections of dog and cat photos. A program could learn to perform this task by observing pictures that have already been sorted, and it could evaluate its performance by calculating the percentage of correctly classified pictures.

We will use Mitchell's definition of machine learning to organize this chapter. First, we will discuss types of experience, including **supervised** learning and **unsupervised** learning. Next, we will discuss common tasks that can be performed by machine learning systems. Finally, we will discuss performance measures that can be used to assess machine learning systems.

Learning from experience

Machine learning systems are often described as learning from experience either with or without supervision from humans. In supervised learning problems, a program predicts an output for an input by learning from pairs of labeled inputs and outputs; that is, the program learns from examples of the right answers. In unsupervised learning, a program does not learn from labeled data. Instead, it attempts to discover patterns in the data. For example, assume that you have collected data describing the heights and weights of people. An example of an unsupervised learning problem is dividing the data points into groups. A program might produce groups that correspond to men and women, or children and adults.

Now assume that the data is also labeled with the person's sex. An example of a supervised learning problem is inducing a rule to predict whether a person is male or female based on his or her height and weight. We will discuss algorithms and examples of supervised and unsupervised learning in the following chapters.

Supervised learning and unsupervised learning can be thought of as occupying opposite ends of a spectrum. Some types of problems, called **semi-supervised** learning

problems, make use of both supervised and unsupervised data; these problems are located on the spectrum between supervised and unsupervised learning. An example of semi-supervised machine learning is reinforcement learning, in which a program receives feedback for its decisions, but the feedback may not be associated with a single decision. For example, a reinforcement learning program that learns to play a side-scrolling video game such as *Super Mario Bros.* may receive a reward when it completes a level or exceeds a certain score, and a punishment when it loses a life. However, this supervised feedback is not associated with specific decisions to run, avoid Goombas, or pick up fire flowers. While this book will discuss semi-supervised learning, we will focus primarily on supervised and unsupervised learning, as these categories include most the common machine learning problems. In the next sections, we will review supervised and unsupervised learning in more detail.

A supervised learning program learns from labeled examples of the outputs that should be produced for an input. There are many names for the output of a machine learning program. Several disciplines converge in machine learning, and many of those disciplines use their own terminology. In this book, we will refer to the output as the **response variable**. Other names for response variables include dependent variables, regressands,

criterion variables, measured variables, responding variables, explained variables, outcome variables, experimental variables, labels, and output variables. Similarly, the input variables have several names. In this book, we will refer to the input variables as **features**, and the phenomena they measure as **explanatory variables**. Other names for explanatory variables include predictors, regressors, controlled variables, manipulated variables, and exposure variables. Response variables and explanatory variables may take real or discrete values.

The collection of examples that comprise supervised experience is called a **training set**. A collection of examples that is used to assess the performance of a program is called a **test set**. The response variable can be thought of as the answer to the question posed by the explanatory variables. Supervised learning problems learn from a collection of answers to different questions; that is, supervised learning programs are provided with the correct answers and must learn to respond correctly to unseen, but similar, questions.

Machine learning tasks

Two of the most common supervised machine learning tasks are **classification** and **regression**. In classification tasks the program must learn to predict discrete values for the response variables from one or more explanatory variables. That is, the program must predict the most probable category, class, or label for new observations. Applications of classification include predicting whether a stock's price will rise or fall, or deciding if a news article belongs to the politics or leisure section. In regression problems the program must predict the value of a continuous response variable. Examples of regression problems include predicting the sales for a new product, or the salary for a job based on its description. Similar to classification, regression problems require supervised learning.

A common unsupervised learning task is to discover groups of related observations, called **clusters**, within the training data. This task, called **clustering** or cluster analysis, assigns observations to groups such that observations within groups are more similar to each other based on some similarity measure than they are to observations in other groups. Clustering is often used to explore a dataset. For example, given a collection of

movie reviews, a clustering algorithm might discover sets of positive and negative reviews. The system will not be able to label the clusters as "positive" or "negative"; without supervision, it will only have knowledge that the grouped observations are similar to each other by some measure. A common application of clustering is discovering segments of customers within a market for a product. By understanding what attributes are common to particular groups of customers, marketers can decide what aspects of their campaigns need to be emphasized. Clustering is also used by Internet radio services; for example, given a collection of songs, a clustering algorithm might be able to group the songs according to their genres. Using different similarity measures, the same clustering algorithm might group the songs by their keys, or by the instruments they contain.

Dimensionality reduction is another common unsupervised learning task. Some problems may contain thousands or even millions of explanatory variables, which can be computationally costly to work with. Additionally, the program's ability to generalize may be reduced if some of the explanatory variables capture noise or are irrelevant to the underlying relationship.

Dimensionality reduction is the process of discovering the explanatory variables that account for the greatest changes in the response variable. Dimensionality reduction can

also be used to visualize data. It is easy to visualize a regression problem such as predicting the price of a home from its size; the size of the home can be plotted on the graph's x axis, and the price of the home can be plotted on the y axis. Similarly, it is easy to visualize the housing price regression problem when a second explanatory variable is added. The number of bathrooms in the house could be plotted on the z axis, for instance. A problem with thousands of explanatory variables, however, becomes impossible to visualize.

Training data and test data

The observations in the training set comprise the experience that the algorithm uses to learn. In supervised learning problems, each observation consists of an observed response variable and one or more observed explanatory variables.

The test set is a similar collection of observations that is used to evaluate the performance of the model using some performance metric. It is important that no observations from the training set are included in the test set. If the test set does contain examples from the training set, it will be difficult to assess whether the algorithm has learned to generalize from the training set or has simply memorized it. A program that generalizes well will be able to effectively perform a task with new data. In contrast, a program that memorizes the training data by learning an overly complex model could predict the values of the response variable for the training set accurately, but will fail to predict the value of the response variable for new examples.

Memorizing the training set is called **over-fitting**. A program that memorizes its observations may not perform its task well, as it could memorize relations and structures

that are noise or coincidence. Balancing memorization and generalization, or over-fitting and under-fitting, is a problem common to many machine learning algorithms. In later chapters we will discuss regularization, which can be applied to many models to reduce over-fitting.

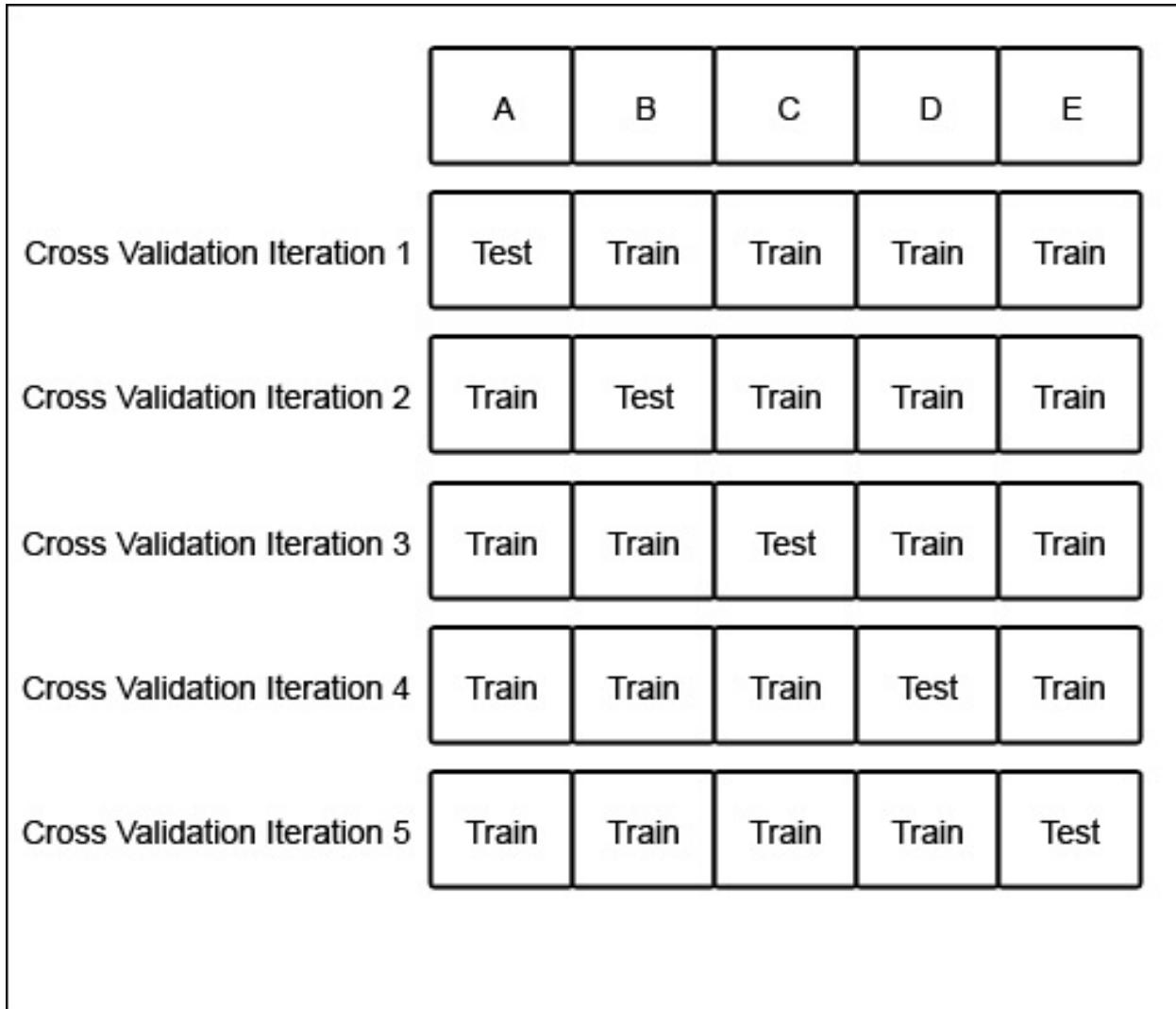
In addition to the training and test data, a third set of observations, called a **validation** or **hold-out** set, is sometimes required. The validation set is used to tune variables called **hyperparameters**, which control how the model is learned. The program is still evaluated on the test set to provide an estimate of its performance in the real world; its performance on the validation set should not be used as an estimate of the model's real-world performance since the program has been tuned specifically to the validation data. It is common to partition a single set of supervised observations into training, validation, and test sets. There are no requirements for the sizes of the partitions, and they may vary according to the amount of data available. It is common to allocate 50 percent or more of the data to the training set, 25 percent to the test set, and the remainder to the validation set.

Some training sets may contain only a few hundred observations; others may include millions. Inexpensive storage, increased network connectivity, the ubiquity of sensor-packed smartphones, and shifting attitudes towards

privacy have contributed to the contemporary state of big data, or training sets with millions or billions of examples. While this book will not work with datasets that require parallel processing on tens or hundreds of machines, the predictive power of many machine learning algorithms improves as the amount of training data increases. However, machine learning algorithms also follow the maxim "garbage in, garbage out." A student who studies for a test by reading a large, confusing textbook that contains many errors will likely not score better than a student who reads a short but well-written textbook. Similarly, an algorithm trained on a large collection of noisy, irrelevant, or incorrectly labeled data will not perform better than an algorithm trained on a smaller set of data that is more representative of problems in the real world.

Many supervised training sets are prepared manually, or by semi-automated processes. Creating a large collection of supervised data can be costly in some domains. Fortunately, several datasets are bundled with scikit-learn, allowing developers to focus on experimenting with models instead. During development, and particularly when training data is scarce, a practice called **cross-validation** can be used to train and validate an algorithm on the same data. In cross-validation, the training data is partitioned. The algorithm is trained using all but one of

the partitions, and tested on the remaining partition. The partitions are then rotated several times so that the algorithm is trained and evaluated on all of the data. The following diagram depicts cross-validation with five partitions or **folds**:



The original dataset is partitioned into five subsets of

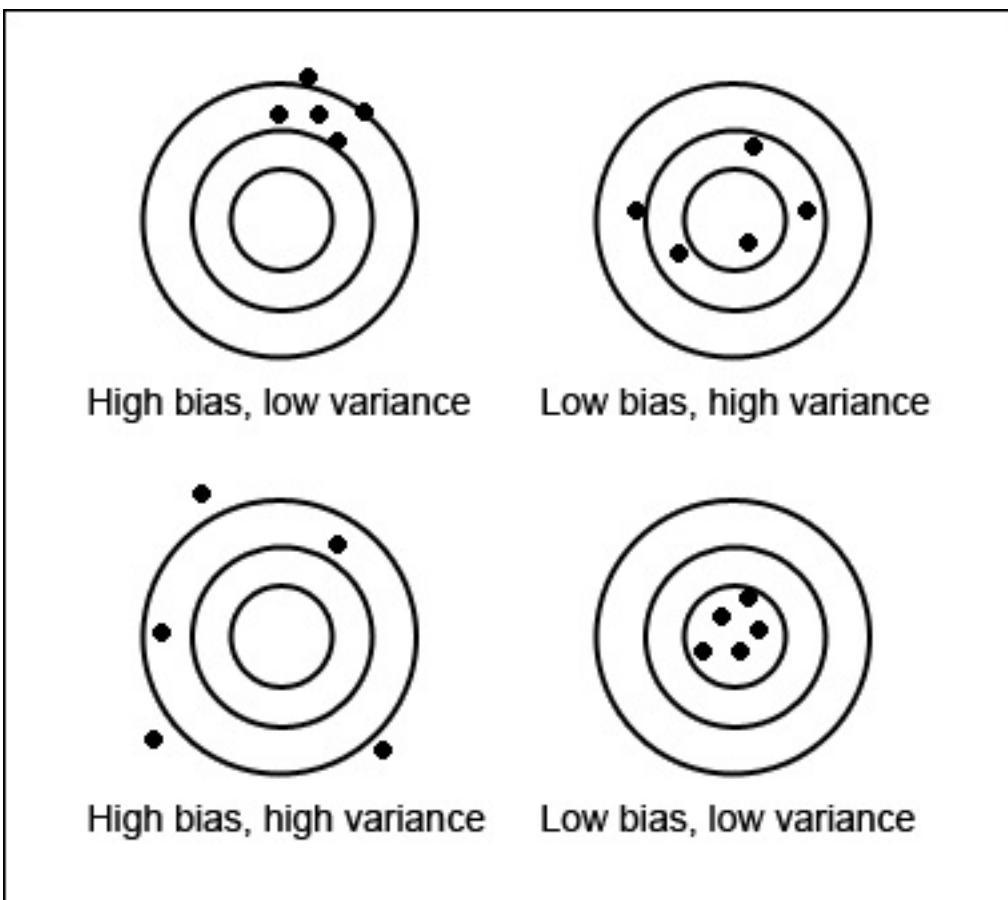
equal size, labeled **A** through **E**. Initially, the model is trained on partitions **B** through **E**, and tested on partition **A**. In the next iteration, the model is trained on partitions **A**, **C**, **D**, and **E**, and tested on partition **B**. The partitions are rotated until models have been trained and tested on all of the partitions. Cross-validation provides a more accurate estimate of the model's performance than testing a single partition of the data.

Performance measures, bias, and variance

Many metrics can be used to measure whether or not a program is learning to perform its task more effectively. For supervised learning problems, many performance metrics measure the number of prediction errors. There are two fundamental causes of prediction error: a model's **bias** and its **variance**. Assume that you have many training sets that are all unique, but equally representative of the population. A model with a high bias will produce similar errors for an input regardless of the training set it was trained with; the model biases its own assumptions about the real relationship over the relationship demonstrated in the training data. A model with high variance, conversely, will produce different errors for an input depending on the training set that it was trained with. A model with high bias is inflexible, but a model with high variance may be so flexible that it models the noise in the training set. That is, a model with high variance over-fits the training data, while a model with high bias under-fits the training data. It can be helpful to visualize bias and variance as darts thrown at a dartboard. Each dart is analogous to a prediction from a different dataset. A model with high bias but low variance will

throw darts that are far from the bull's eye, but tightly clustered. A model with high bias and high variance will throw darts all over the board; the darts are far from the bull's eye and each other.

A model with low bias and high variance will throw darts that are closer to the bull's eye, but poorly clustered. Finally, a model with low bias and low variance will throw darts that are tightly clustered around the bull's eye, as shown in the following diagram:



Ideally, a model will have both low bias and variance, but efforts to decrease one will frequently increase the other. This is known as the **bias-variance trade-off**. We will discuss the biases and variances of many of the models introduced in this book.

Unsupervised learning problems do not have an error signal to measure; instead, performance metrics for unsupervised learning problems measure some attributes of the structure discovered in the data.

Most performance measures can only be calculated for a specific type of task. Machine learning systems should be evaluated using performance measures that represent the costs associated with making errors in the real world. While this may seem obvious, the following example describes the use of a performance measure that is appropriate for the task in general but not for its specific application.

Consider a classification task in which a machine learning system observes tumors and must predict whether these tumors are malignant or benign. **Accuracy**, or the fraction of instances that were classified correctly, is an intuitive measure of the program's performance. While accuracy does measure the program's performance, it does not differentiate between malignant tumors that were

classified as being benign, and benign tumors that were classified as being malignant. In some applications, the costs associated with all types of errors may be the same. In this problem, however, failing to identify malignant tumors is likely to be a more severe error than mistakenly classifying benign tumors as being malignant.

We can measure each of the possible prediction outcomes to create different views of the classifier's performance. When the system correctly classifies a tumor as being malignant, the prediction is called a **true positive**. When the system incorrectly classifies a benign tumor as being malignant, the prediction is a **false positive**. Similarly, a **false negative** is an incorrect prediction that the tumor is benign, and a **true negative** is a correct prediction that a tumor is benign. These four outcomes can be used to calculate several common measures of classification performance, including **accuracy**, **precision**, and **recall**.

Accuracy is calculated with the following formula, where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision is the fraction of the tumors that were predicted to be malignant that are actually malignant. Precision is calculated with the following formula:

$$P = \frac{TP}{TP + FP}$$

Recall is the fraction of malignant tumors that the system identified. Recall is calculated with the following formula:

$$R = \frac{TP}{TP + FN}$$

In this example, precision measures the fraction of tumors that were predicted to be malignant that are actually malignant. Recall measures the fraction of truly malignant tumors that were detected.

The precision and recall measures could reveal that a classifier with impressive accuracy actually fails to detect most of the malignant tumors. If most tumors are benign,

even a classifier that never predicts malignancy could have high accuracy. A different classifier with lower accuracy and higher recall might be better suited to the task, since it will detect more of the malignant tumors.

Many other performance measures for classification can be used; we will discuss some, including metrics for multilabel classification problems, in later chapters. In the next chapter, we will discuss some common performance measures for regression tasks.

An introduction to scikit-learn

Since its release in 2007, scikit-learn has become one of the most popular open source machine learning libraries for Python. scikit-learn provides algorithms for machine learning tasks including classification, regression, dimensionality reduction, and clustering. It also provides modules for extracting features, processing data, and evaluating models.

Conceived as an extension to the SciPy library, scikit-learn is built on the popular Python libraries NumPy and matplotlib. NumPy extends Python to support efficient operations on large arrays and multidimensional matrices. matplotlib provides visualization tools, and SciPy provides modules for scientific computing.

scikit-learn is popular for academic research because it has a well-documented, easy-to-use, and versatile API. Developers can use scikit-learn to experiment with different algorithms by changing only a few lines of the code. scikit-learn wraps some popular implementations of machine learning algorithms, such as LIBSVM and LIBLINEAR. Other Python libraries, including NLTK,

include wrappers for scikit-learn. scikit-learn also includes a variety of datasets, allowing developers to focus on algorithms rather than obtaining and cleaning data.

Licensed under the permissive BSD license, scikit-learn can be used in commercial applications without restrictions. Many of scikit-learn's algorithms are fast and scalable to all but massive datasets. Finally, scikit-learn is noted for its reliability; much of the library is covered by automated tests.

Installing scikit-learn

This book is written for version 0.15.1 of scikit-learn; use this version to ensure that the examples run correctly. If you have previously installed scikit-learn, you can retrieve the version number with the following code:

```
>>> import sklearn  
>>> sklearn.__version__  
'0.15.1'
```

If you have not previously installed scikit-learn, you can install it from a package manager or build it from the source. We will review the installation processes for Linux, OS X, and Windows in the following sections, but refer to <http://scikit-learn.org/stable/install.html> for the latest instructions. The following instructions only assume that you have installed Python 2.6, Python 2.7, or Python 3.2 or newer. Go to <http://www.python.org/download/> for instructions on how to install Python.

Installing scikit-learn on Windows

scikit-learn requires Setuptools, a third-party package that supports packaging and installing software for Python. Setuptools can be installed on Windows by running the bootstrap script at

https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py

Windows binaries for the 32- and 64-bit versions of scikit-learn are also available. If you cannot determine which version you need, install the 32-bit version. Both versions depend on NumPy 1.3 or newer. The 32-bit version of NumPy can be downloaded from

<http://sourceforge.net/projects/numpy/files/NumPy/>. The

64-bit version can be downloaded from

<http://www.lfd.uci.edu/~gohlke/pythonlibs/#scikit-learn>.

A Windows installer for the 32-bit version of scikit-learn can be downloaded from

<http://sourceforge.net/projects/scikit-learn/files/>. An

installer for the 64-bit version of scikit-learn can be downloaded from

<http://www.lfd.uci.edu/~gohlke/pythonlibs/#scikit-learn>.

scikit-learn can also be built from the source code on

Windows. Building requires a C/C++ compiler such as MinGW (<http://www.mingw.org/>), NumPy, SciPy, and Setuptools.

To build, clone the Git repository from <https://github.com/scikit-learn/scikit-learn> and execute the following command:

```
python setup.py install
```

Installing scikit-learn on Linux

There are several options to install scikit-learn on Linux, depending on your distribution. The preferred option to install scikit-learn on Linux is to use `pip`. You may also install it using a package manager, or build scikit-learn from its source.

To install scikit-learn using `pip`, execute the following command:

```
sudo pip install scikit-learn
```

To build scikit-learn, clone the Git repository from <https://github.com/scikit-learn/scikit-learn>. Then install the following dependencies:

```
sudo apt-get install python-dev python-numpy  
python-numpy-dev python-setuptools python-numpy-  
dev python-scipy libatlas-dev g++
```

Navigate to the repository's directory and execute the following command:

```
python setup.py install
```

Installing scikit-learn on OS X

scikit-learn can be installed on OS X using Macports:

```
sudo port install py26-sklearn
```

If Python 2.7 is installed, run the following command:

```
sudo port install py27-sklearn
```

scikit-learn can also be installed using pip with the following command:

```
pip install scikit-learn
```

Verifying the installation

To verify that scikit-learn has been installed correctly, open a Python console and execute the following:

```
>>> import sklearn  
>>> sklearn.__version__  
'0.15.1'
```

To run scikit-learn's unit tests, first install the `nose` library. Then execute the following:

```
nosetests sklearn -exe
```

Congratulations! You've successfully installed scikit-learn.

Installing pandas and matplotlib

pandas is an open source library that provides data structures and analysis tools for Python. pandas is a powerful library, and several books describe how to use pandas for data analysis. We will use a few of panda's convenient tools for importing data and calculating summary statistics.

pandas can be installed on Windows, OS X, and Linux using `pip` with the following command:

```
pip install pandas
```

pandas can also be installed on Debian- and Ubuntu-based Linux distributions using the following command:

```
apt-get install python-pandas
```

matplotlib is a library used to easily create plots, histograms, and other charts with Python. We will use it to visualize training data and models. matplotlib has several dependencies. Like pandas, matplotlib depends on NumPy, which should already be installed. On Debian- and Ubuntu-based Linux distributions, matplotlib and its

dependencies can be installed using the following command:

```
apt-get install python-matplotlib
```

Binaries for OS X and Windows can be downloaded from
<http://matplotlib.org/downloads.html>.

Summary

In this chapter we defined machine-learning as the design and study of programs that can improve their performance of a task by learning from experience. We discussed the spectrum of supervision in experience. At one end of the spectrum is supervised learning, in which a program learns from inputs that are labeled with their corresponding outputs. At the opposite end of the spectrum is unsupervised learning, in which the program must discover hidden structure in unlabeled data. Semi-supervised approaches make use of both labeled and unlabeled training data.

We discussed common types of machine learning tasks and reviewed example applications. In classification tasks the program must predict the value of a discrete response variable from the explanatory variables. In regression tasks the program must predict the value of a continuous response variable from the explanatory variables. In regression tasks, the program must predict the value of a continuous response variable from the explanatory variables. Unsupervised learning tasks include clustering, in which observations are organized into groups according to some similarity measure and dimensionality reduction, which reduces a set of explanatory variables to

a smaller set of synthetic features that retain as much information as possible. We also reviewed the bias-variance trade-off and discussed common performance measures for different machine learning tasks.

We also discussed the history, goals, and advantages of scikit-learn. Finally, we prepared our development environment by installing scikit-learn and other libraries that are commonly used in conjunction with it. In the next chapter, we will discuss the regression task in more detail, and build our first machine learning model with scikit-learn.

Chapter 2. Linear Regression

In this chapter you will learn how to use linear models in regression problems. First, we will examine simple linear regression, which models the relationship between a response variable and single explanatory variable. Next, we will discuss multiple linear regression, a generalization of simple linear regression that can support more than one explanatory variable. Then, we will discuss polynomial regression, a special case of multiple linear regression that can effectively model nonlinear relationships. Finally, we will discuss how to train our models by finding the values of their parameters that minimize a cost function. We will work through a toy problem to learn how the models and learning algorithms work before discussing an application with a larger dataset.

Simple linear regression

In the previous chapter you learned that training data is used to estimate the parameters of a model in supervised learning problems. Past observations of explanatory variables and their corresponding response variables comprise the training data. The model can be used to predict the value of the response variable for values of the explanatory variable that have not been previously observed. Recall that the goal in regression problems is to predict the value of a continuous response variable. In this chapter, we will examine several example linear regression models. We will discuss the training data, model, learning algorithm, and evaluation metrics for each approach. To start, let's consider **simple linear regression**. Simple linear regression can be used to model a linear relationship between one response variable and one explanatory variable. Linear regression has been applied to many important scientific and social problems; the example that we will consider is probably not one of them.

Suppose you wish to know the price of a pizza. You might simply look at a menu. This, however, is a machine learning book, so we will use simple linear regression instead to predict the price of a pizza based on an attribute

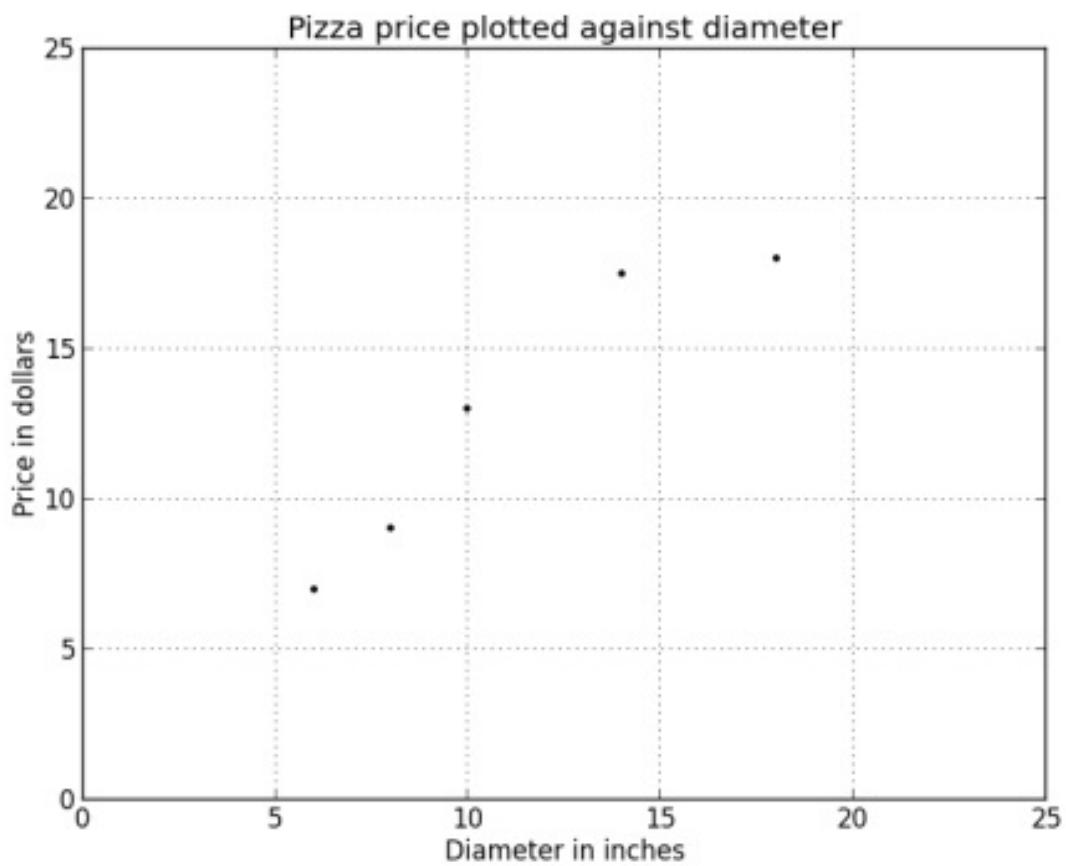
of the pizza that we can observe. Let's model the relationship between the size of a pizza and its price. First, we will write a program with scikit-learn that can predict the price of a pizza given its size. Then, we will discuss how simple linear regression works and how it can be generalized to work with other types of problems. Let's assume that you have recorded the diameters and prices of pizzas that you have previously eaten in your pizza journal. These observations comprise our training data:

Training instance	Diameter (in inches)	Price (in dollars)
1	6	7
2	8	9
3	10	13
4	14	17.5
5	18	18

We can visualize our training data by plotting it on a graph using matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> X = [[6], [8], [10], [14], [18]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> plt.figure()
>>> plt.title('Pizza price plotted against
diameter')
>>> plt.xlabel('Diameter in inches')
>>> plt.ylabel('Price in dollars')
>>> plt.plot(X, y, 'k.')
>>> plt.axis([0, 25, 0, 25])
>>> plt.grid(True)
>>> plt.show()
```

The preceding script produces the following graph. The diameters of the pizzas are plotted on the x axis and the prices are plotted on the y axis.



We can see from the graph of the training data that there is a positive relationship between the diameter of a pizza and its price, which should be corroborated by our own pizza-eating experience. As the diameter of a pizza increases, its price generally increases too. The following pizza-price predictor program models this relationship using linear regression. Let's review the following program and discuss how linear regression works:

```
>>> from sklearn.linear_model import  
LinearRegression  
>>> # Training data  
>>> X = [[6], [8], [10], [14], [18]]  
>>> y = [[7], [9], [13], [17.5], [18]]  
>>> # Create and fit the model  
>>> model = LinearRegression()  
>>> model.fit(X, y)  
>>> print 'A 12" pizza should cost: $%.2f' %  
model.predict([12])[0]  
A 12" pizza should cost: $13.68
```

Simple linear regression assumes that a linear relationship exists between the response variable and explanatory variable; it models this relationship with a linear surface called a hyperplane. A hyperplane is a subspace that has one dimension less than the ambient space that contains it. In simple linear regression, there is one dimension for the response variable and another dimension for the explanatory variable, making a total of two dimensions. The regression hyperplane therefore, has one dimension; a hyperplane with one dimension is a line.

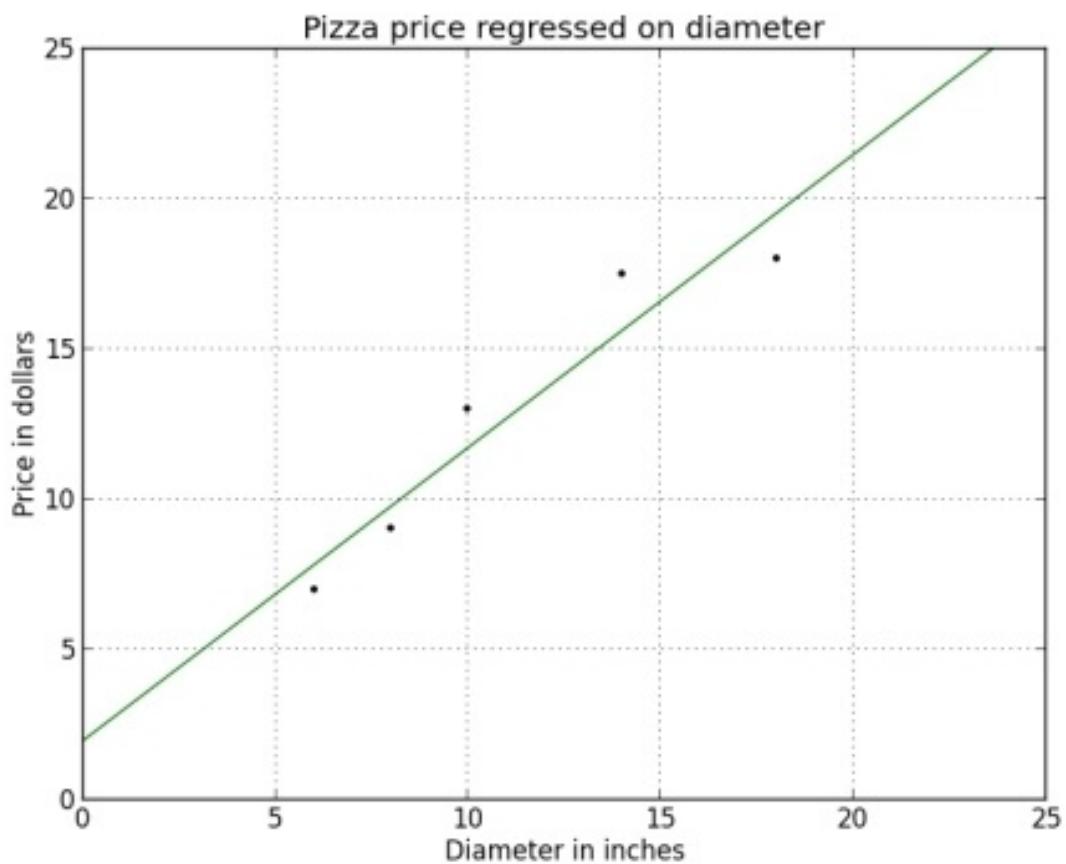
The `sklearn.linear_model.LinearRegression` class is an **estimator**. Estimators predict a value based on the observed data. In scikit-learn, all estimators implement the `fit()` and `predict()` methods. The former method is used to learn the parameters of a model, and the latter method is used to predict the value of a response variable

for an explanatory variable using the learned parameters. It is easy to experiment with different models using scikit-learn because all estimators implement the `fit` and `predict` methods.

The `fit` method of `LinearRegression` learns the parameters of the following model for simple linear regression:

$$y = \alpha + \beta x$$

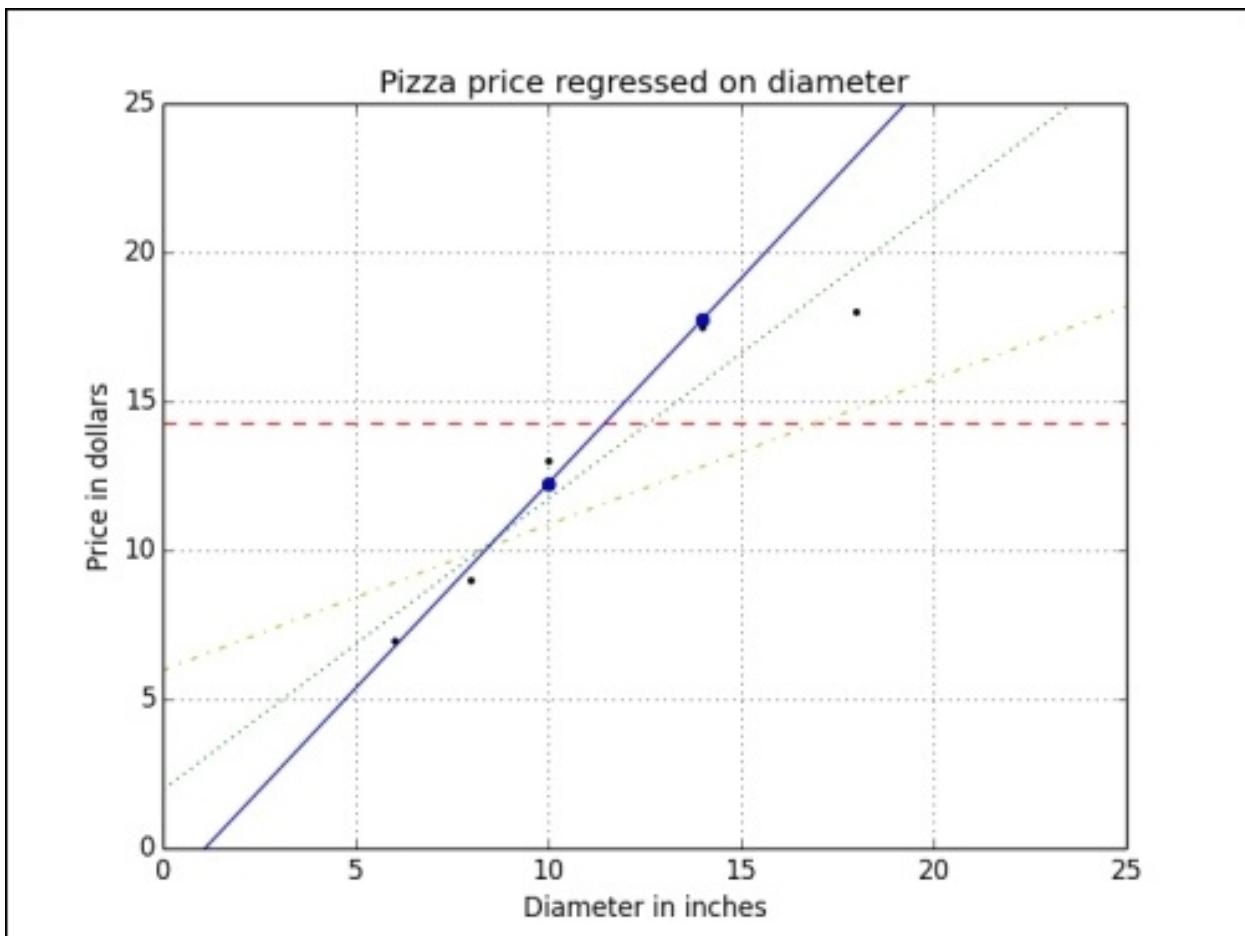
y is the predicted value of the response variable; in this example, it is the predicted price of the pizza. x is the explanatory variable. The intercept term α and coefficient β are parameters of the model that are learned by the learning algorithm. The line plotted in the following figure models the relationship between the size of a pizza and its price. Using this model, we would expect the price of an 8-inch pizza to be about \$7.33, and the price of a 20-inch pizza to be \$18.75.



Using training data to learn the values of the parameters for simple linear regression that produce the best fitting model is called **ordinary least squares** or **linear least squares**. "In this chapter we will discuss methods for approximating the values of the model's parameters and for solving them analytically. First, however, we must define what it means for a model to fit the training data.

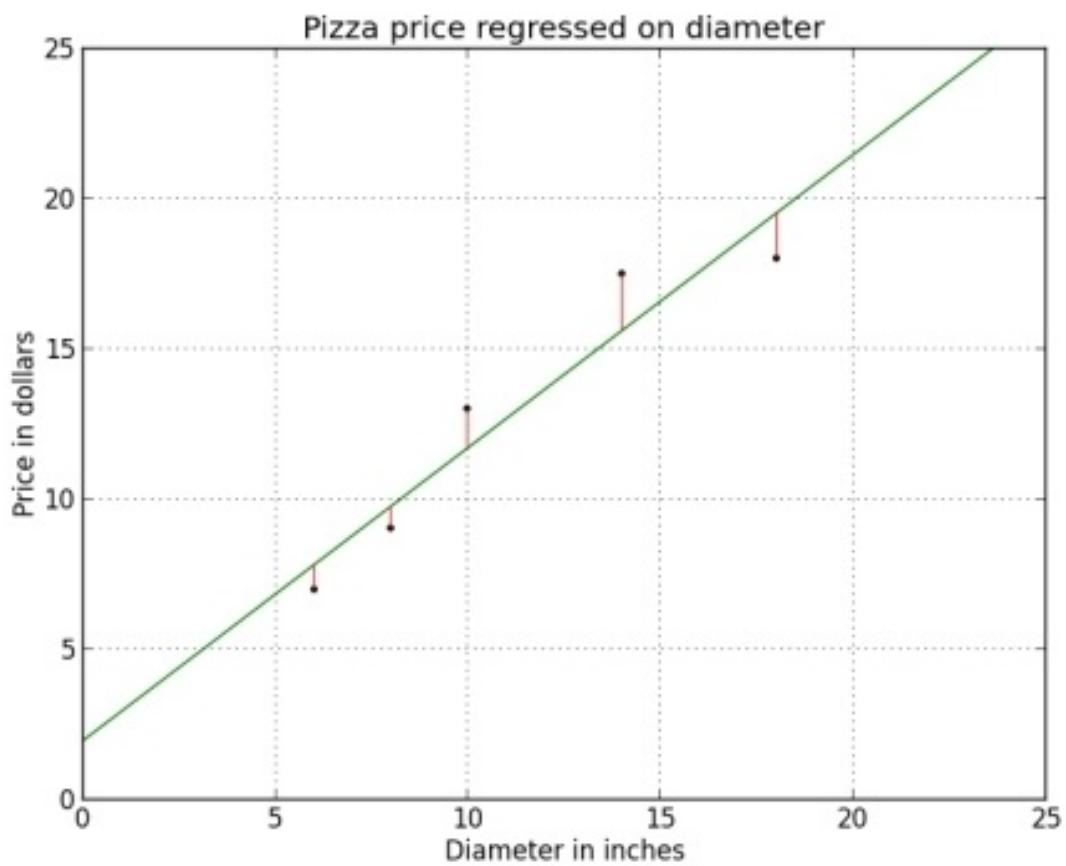
Evaluating the fitness of a model with a cost function

Regression lines produced by several sets of parameter values are plotted in the following figure. How can we assess which parameters produced the best-fitting regression line?



A **cost function**, also called a **loss function**, is used to define and measure the error of a model. The differences between the prices predicted by the model and the observed prices of the pizzas in the training set are called **residuals** or **training errors**. Later, we will evaluate a model on a separate set of test data; the differences between the predicted and observed values in the test data are called **prediction errors** or **test errors**.

The residuals for our model are indicated by the vertical lines between the points for the training instances and regression hyperplane in the following plot:



We can produce the best pizza-price predictor by minimizing the sum of the residuals. That is, our model fits if the values it predicts for the response variable are close to the observed values for all of the training examples. This measure of the model's fitness is called the **residual sum of squares** cost function. Formally, this function assesses the fitness of a model by summing the squared residuals for all of our training examples. The

residual sum of squares is calculated with the formula in the following equation, where y_i is the observed value and $f(x_i)$ is the predicted value:

$$SS_{res} = \sum_{i=1}^n (y_i - f(x_i))^2$$

Let's compute the residual sum of squares for our model by adding the following two lines to the previous script:

```
>>> import numpy as np
>>> print 'Residual sum of squares: %.2f' %
np.mean((model.predict(X) - y) ** 2)
Residual sum of squares: 1.75
```

Now that we have a cost function, we can find the values of our model's parameters that minimize it.

Solving ordinary least squares for simple linear regression

In this section, we will work through solving ordinary least squares for simple linear regression. Recall that simple linear regression is given by the following equation:

$$y = \alpha + \beta x$$

Also, recall that our goal is to solve the values of β and α that minimize the cost function. We will solve β first. To do so, we will calculate the **variance** of x and **covariance** of x and y .

Variance is a measure of how far a set of values is spread out. If all of the numbers in the set are equal, the variance of the set is zero. A small variance indicates that the numbers are near the mean of the set, while a set containing numbers that are far from the mean and each other will have a large variance. Variance can be calculated using the following equation:

$$\text{var}(x) = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}$$

In the preceding equation, \bar{x} is the mean of x , x_i is the value of x for the i th training instance, and n is the number of training instances. Let's calculate the variance of the pizza diameters in our training set:

```
>>> from __future__ import division
>>> xbar = (6 + 8 + 10 + 14 + 18) / 5
>>> variance = ((6 - xbar)**2 + (8 - xbar)**2 +
(10 - xbar)**2 + (14 - xbar)**2 + (18 -
xbar)**2) / 4
>>> print variance
23.2
```

NumPy also provides the `var` method to calculate variance. The `ddof` keyword parameter can be used to set Bessel's correction to calculate the sample variance:

```
>>> import numpy as np
>>> print np.var([6, 8, 10, 14, 18], ddof=1)
23.2
```

Covariance is a measure of how much two variables change together. If the value of the variables increase together, their covariance is positive. If one variable tends

to increase while the other decreases, their covariance is negative. If there is no linear relationship between the two variables, their covariance will be equal to zero; the variables are linearly uncorrelated but not necessarily independent. Covariance can be calculated using the following formula:

$$\text{cov}(x, y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{n-1}$$

As with variance, x_i is the diameter of the i th training instance, \bar{x} is the mean of the diameters, \bar{y} is the mean of the prices, y_i is the price of the i th training instance, and n is the number of training instances. Let's calculate the covariance of the diameters and prices of the pizzas in the training set:

```
>>> xbar = (6 + 8 + 10 + 14 + 18) / 5
>>> ybar = (7 + 9 + 13 + 17.5 + 18) / 5
>>> cov = ((6 - xbar) * (7 - ybar) + (8 - xbar)
* (9 - ybar) + (10 - xbar) * (13 - ybar) +
>>>           (14 - xbar) * (17.5 - ybar) + (18 -
xbar) * (18 - ybar)) / 4
>>> print cov
>>> import numpy as np
>>> print np.cov([6, 8, 10, 14, 18], [7, 9, 13,
17.5, 18])[0][1]
```

22.65

22.65

Now that we have calculated the variance of our explanatory variable and the covariance of the response and explanatory variables, we can solve β using the following formula:

$$\beta = \frac{\text{cov}(x, y)}{\text{var}(x)}$$

$$\beta = \frac{22.65}{23.2} = 0.9762931034482758$$

Having solved β , we can solve α using the following formula:

$$\alpha = \bar{y} - \beta \bar{x}$$

In the preceding formula, \bar{y} is the mean of y and \bar{x} is the mean of x . (\bar{x}, \bar{y}) are the coordinates of the centroid, a point that the model must pass through. We can use the

centroid and the value of β to solve for α as follows:

$$\alpha = 12.9 - 0.9762931034482758 \times 11.2 = 1.9655172413793114$$

Now that we have solved the values of the model's parameters that minimize the cost function, we can plug in the diameters of the pizzas and predict their prices. For instance, an 11-inch pizza is expected to cost around \$12.70, and an 18-inch pizza is expected to cost around \$19.54. Congratulations! You used simple linear regression to predict the price of a pizza.

Evaluating the model

We have used a learning algorithm to estimate a model's parameters from the training data. How can we assess whether our model is a good representation of the real relationship? Let's assume that you have found another page in your pizza journal. We will use the entries on this page as a test set to measure the performance of our model:

Test Instance	Diameter (in inches)	Observed price (in dollars)	Predicted price (in dollars)
1	8	11	9.7759
2	9	8.5	10.7522
3	11	15	12.7048
4	16	18	17.5863
5	12	11	13.6811

Several measures can be used to assess our model's predictive capabilities. We will evaluate our pizza-price

predictor using **r-squared**. R-squared measures how well the observed values of the response variables are predicted by the model. More concretely, r-squared is the proportion of the variance in the response variable that is explained by the model. An r-squared score of one indicates that the response variable can be predicted without any error using the model. An r-squared score of one half indicates that half of the variance in the response variable can be predicted using the model. There are several methods to calculate r-squared. In the case of simple linear regression, r-squared is equal to the square of the Pearson product moment correlation coefficient, or Pearson's r .

Using this method, r-squared must be a positive number between zero and one. This method is intuitive; if r-squared describes the proportion of variance in the response variable explained by the model, it cannot be greater than one or less than zero. Other methods, including the method used by scikit-learn, do not calculate r-squared as the square of Pearson's r , and can return a negative r-squared if the model performs extremely poorly. We will follow the method used by scikit-learn to calculate r-squared for our pizza-price predictor.

First, we must measure the total sum of the squares. Σ is

the observed value of the response variable for the i th test instance, and \bar{y} is the mean of the observed values of the response variable

$$SS_{tot} = \sum_{i=1}^n (y_i - \bar{y})^2$$

$$SS_{tot} = (11 - 12.7)^2 + (8.5 - 12.7)^2 + \dots + (11 - 12.7)^2 = 56.8$$

Next, we must find the residual sum of the squares. Recall that this is also our cost function.

$$SS_{res} = \sum_{i=1}^n (y_i - f(x_i))^2$$

$$SS_{res} = (11 - 9.7759)^2 + (8.5 - 10.7522)^2 + \dots + (11 - 13.6811)^2 = 19.19821359$$

Finally, we can find r-squared using the following formula:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

$$R^2 = 1 - \frac{19.19821359}{56.8} = 0.6620032818661972$$

An r-squared score of **0.6620** indicates that a large proportion of the variance in the test instances' prices is explained by the model. Now, let's confirm our calculation using scikit-learn. The `score` method of `LinearRegression` returns the model's r-squared value, as seen in the following example:

```
>>> from sklearn.linear_model import  
LinearRegression  
>>> X = [[6], [8], [10], [14], [18]]  
>>> y = [[7], [9], [13], [17.5], [18]]  
>>> X_test = [[8], [9], [11], [16], [12]]  
>>> y_test = [[11], [8.5], [15], [18], [11]]  
>>> model = LinearRegression()  
>>> model.fit(X, y)  
>>> print 'R-squared: %.4f' %  
model.score(X_test, y_test)  
R-squared: 0.6620
```

Multiple linear regression

We have trained and evaluated a model to predict the price of a pizza. While you are eager to demonstrate the pizza-price predictor to your friends and co-workers, you are concerned by the model's imperfect r-squared score and the embarrassment its predictions could cause you. How can we improve the model?

Recalling your personal pizza-eating experience, you might have some intuitions about the other attributes of a pizza that are related to its price. For instance, the price often depends on the number of toppings on the pizza. Fortunately, your pizza journal describes toppings in detail; let's add the number of toppings to our training data as a second explanatory variable. We cannot proceed with simple linear regression, but we can use a generalization of simple linear regression that can use multiple explanatory variables called multiple linear regression. Formally, multiple linear regression is the following model:

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

Where simple linear regression uses a single explanatory variable with a single coefficient, multiple linear regression uses a coefficient for each of an arbitrary number of explanatory variables.

$$Y = X\beta$$

For simple linear regression, this is equivalent to the following:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} \alpha + \beta X_1 \\ \alpha + \beta X_2 \\ \vdots \\ \alpha + \beta X_n \end{bmatrix} = \begin{bmatrix} 1 & X_1 \\ 1 & X_2 \\ \vdots & \vdots \\ 1 & X_n \end{bmatrix} \times \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Y is a column vector of the values of the response variables for the training examples. β is a column vector of the values of the model's parameters. X , called the design matrix, is an $m \times n$ dimensional matrix of the values of the explanatory variables for the training examples. m is the number of training examples and n is the number of explanatory variables. Let's update our pizza training data to include the number of toppings with the following

values:

Training Example	Diameter (in inches)	Number of toppings	Price (in dollars)
1	6	2	7
2	8	1	9
3	10	0	13
4	14	2	17.5
5	18	0	18

We must also update our test data to include the second explanatory variable, as follows:

Test Instance	Diameter (in inches)	Number of toppings	Price (in dollars)
1	8	2	11
2	9	0	8.5
3	11	2	15

4	16	2	18
5	12	0	11

Our learning algorithm must estimate the values of three parameters: the coefficients for the two features and the intercept term. While one might be tempted to solve β by dividing each side of the equation by X , division by a matrix is impossible. Just as dividing a number by an integer is equivalent to multiplying by the inverse of the same integer, we can multiply β by the inverse of X to avoid matrix division. Matrix inversion is denoted with a superscript -1. Only square matrices can be inverted. X is not likely to be a square; the number of training instances will have to be equal to the number of features for it to be so. We will multiply X by its transpose to yield a square matrix that can be inverted. Denoted with a superscript T , the transpose of a matrix is formed by turning the rows of the matrix into columns and vice versa, as follows:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

To recap, our model is given by the following formula:

$$Y = X\beta$$

We know the values of Y and X from our training data. We must find the values of β , which minimize the cost function. We can solve β as follows:

$$\beta = (X^T X)^{-1} X^T Y$$

We can solve β using NumPy, as follows:

```
>>> from numpy.linalg import inv
>>> from numpy import dot, transpose
>>> X = [[1, 6, 2], [1, 8, 1], [1, 10, 0], [1, 14, 2], [1, 18, 0]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> print dot(inv(dot(transpose(X), X)), dot(transpose(X), y))
[[ 1.1875      ]
 [ 1.01041667]
 [ 0.39583333]]
```

NumPy also provides a least squares function that can solve the values of the parameters more compactly:

```
>>> from numpy.linalg import lstsq
>>> X = [[1, 6, 2], [1, 8, 1], [1, 10, 0], [1,
14, 2], [1, 18, 0]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> print lstsq(X, y)[0]
[[ 1.1875]
 [ 1.01041667]
 [ 0.39583333]]
```

Let's update our pizza-price predictor program to use the second explanatory variable, and compare its performance on the test set to that of the simple linear regression model:

```
>>> from sklearn.linear_model import
LinearRegression
>>> X = [[6, 2], [8, 1], [10, 0], [14, 2], [18,
0]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> model = LinearRegression()
>>> model.fit(X, y)
>>> X_test = [[8, 2], [9, 0], [11, 2], [16, 2],
[12, 0]]
>>> y_test = [[11], [8.5], [15], [18],
[11]]
>>> predictions = model.predict(X_test)
>>> for i, prediction in enumerate(predictions):
>>>     print 'Predicted: %s, Target: %s' %
(prediction, y_test[i])
>>> print 'R-squared: %.2f' %
model.score(X_test, y_test)
Predicted: [ 10.0625], Target: [11]
Predicted: [ 10.28125], Target: [8.5]
```

```
Predicted: [ 13.09375], Target: [15]
Predicted: [ 18.14583333], Target: [18]
Predicted: [ 13.3125], Target: [11]
R-squared: 0.77
```

It appears that adding the number of toppings as an explanatory variable has improved the performance of our model. In later sections, we will discuss why evaluating a model on a single test set can provide inaccurate estimates of the model's performance, and how we can estimate its performance more accurately by training and testing on many partitions of the data. For now, however, we can accept that the multiple linear regression model performs significantly better than the simple linear regression model. There may be other attributes of pizzas that can be used to explain their prices. What if the relationship between these explanatory variables and the response variable is not linear in the real world? In the next section, we will examine a special case of multiple linear regression that can be used to model nonlinear relationships.

Polynomial regression

In the previous examples, we assumed that the real relationship between the explanatory variables and the response variable is linear. This assumption is not always true. In this section, we will use **polynomial regression**, a special case of multiple linear regression that adds terms with degrees greater than one to the model. The real-world curvilinear relationship is captured when you transform the training data by adding polynomial terms, which are then fit in the same manner as in multiple linear regression. For ease of visualization, we will again use only one explanatory variable, the pizza's diameter. Let's compare linear regression with polynomial regression using the following datasets:

Training Instance	Diameter (in inches)	Price (in dollars)
1	6	7
2	8	9
3	10	13
4	14	17.5

5	18	18
<hr/>		
Testing Instance	Diameter (in inches)	Price (in dollars)
1	6	7
2	8	9
3	10	13
4	14	17.5

Quadratic regression, or regression with a second order polynomial, is given by the following formula:

$$y = \alpha + \beta_1 x + \beta_2 x^2$$

We are using only one explanatory variable, but the model now has three terms instead of two. The explanatory variable has been transformed and added as a third term to the model to capture the curvilinear relationship. Also, note that the equation for polynomial regression is the same as the equation for multiple linear

regression in vector notation. The `PolynomialFeatures` transformer can be used to easily add polynomial features to a feature representation. Let's fit a model to these features, and compare it to the simple linear regression model:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.linear_model import
LinearRegression
>>> from sklearn.preprocessing import
PolynomialFeatures

>>> X_train = [[6], [8], [10], [14], [18]]
>>> y_train = [[7], [9], [13], [17.5], [18]]
>>> X_test = [[6], [8], [11], [16]]
>>> y_test = [[8], [12], [15], [18]]

>>> regressor = LinearRegression()
>>> regressor.fit(X_train, y_train)
>>> xx = np.linspace(0, 26, 100)
>>> yy =
regressor.predict(xx.reshape(xx.shape[0], 1))
>>> plt.plot(xx, yy)

>>> quadratic_featurizer =
PolynomialFeatures(degree=2)
>>> X_train_quadratic =
quadratic_featurizer.fit_transform(X_train)
>>> X_test_quadratic =
quadratic_featurizer.transform(X_test)
```

```
>>> regressor_quadratic = LinearRegression()
>>> regressor_quadratic.fit(X_train_quadratic,
y_train)
>>> xx_quadratic =
quadratic_featurizer.transform(xx.reshape(xx.shape[0], 1))

>>> plt.plot(xx,
regressor_quadratic.predict(xx_quadratic),
c='r', linestyle='--')
>>> plt.title('Pizza price regressed on
diameter')
>>> plt.xlabel('Diameter in inches')
>>> plt.ylabel('Price in dollars')
>>> plt.axis([0, 25, 0, 25])
>>> plt.grid(True)
>>> plt.scatter(X_train, y_train)
>>> plt.show()

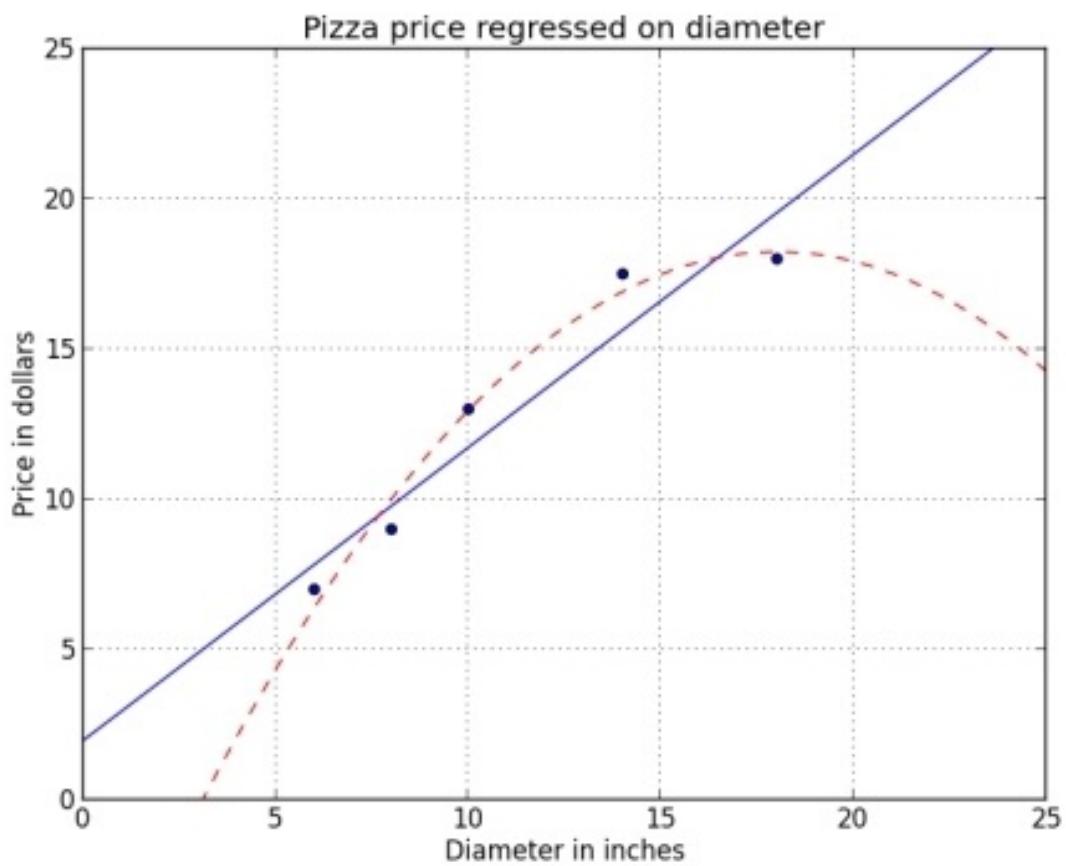
>>> print X_train
>>> print X_train_quadratic
>>> print X_test
>>> print X_test_quadratic
>>> print 'Simple linear regression r-squared',
regressor.score(X_test, y_test)
>>> print 'Quadratic regression r-squared',
regressor_quadratic.score(X_test_quadratic,
y_test)
```

The following is the output of the preceding script:

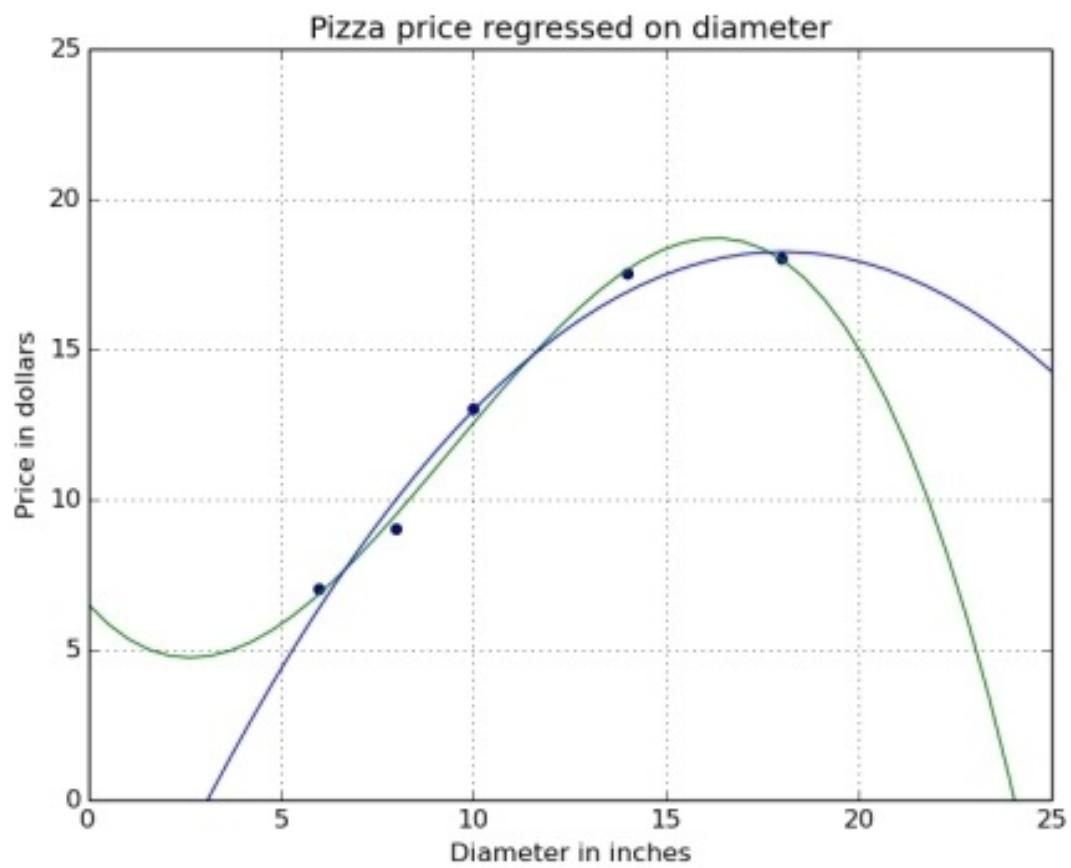
```
[[6], [8], [10], [14], [18]]
[[ 1    6   36]]
```

```
[ 1  8  64]
[ 1 10 100]
[ 1 14 196]
[ 1 18 324]]
[[6], [8], [11], [16]]
[[ 1  6  36]
[ 1  8  64]
[ 1 11 121]
[ 1 16 256]]
Simple linear regression r-squared
0.809726797708
Quadratic regression r-squared 0.867544365635
```

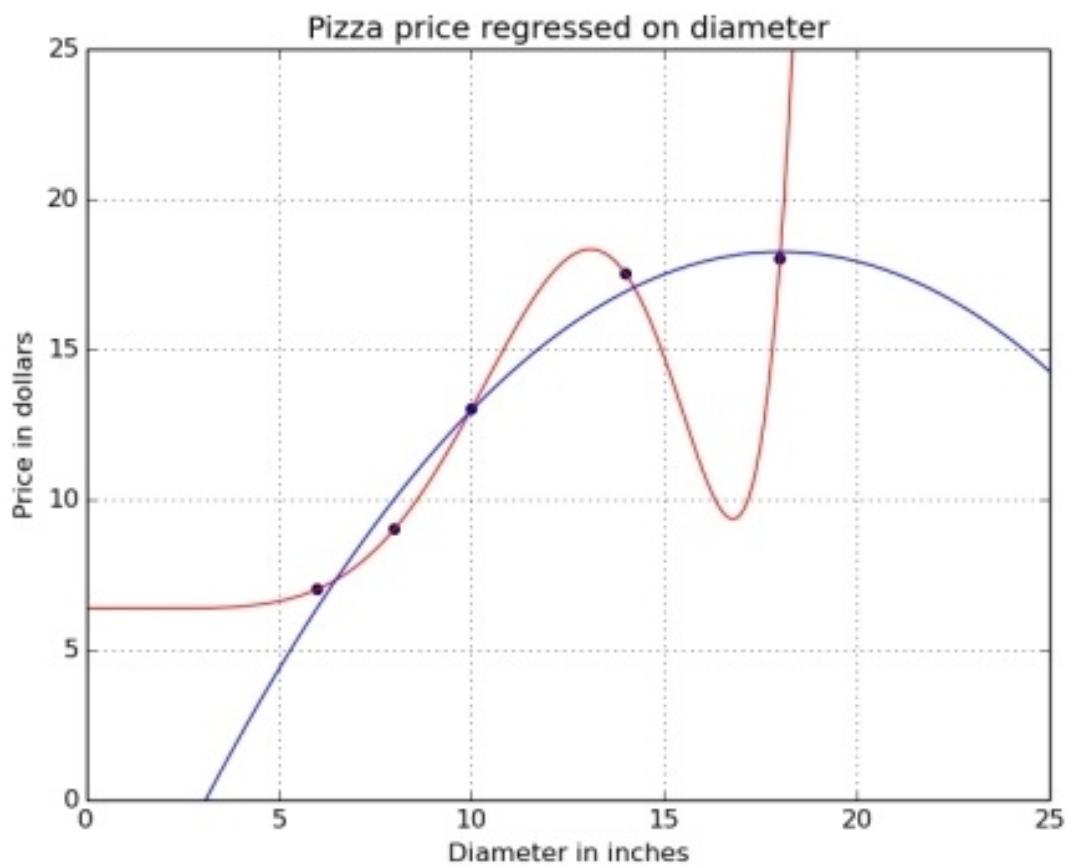
The simple linear regression model is plotted with the solid line in the following figure. Plotted with a dashed line, the quadratic regression model visibly fits the training data better.



The r-squared score of the simple linear regression model is 0.81; the quadratic regression model's r-squared score is an improvement at 0.87. While quadratic and cubic regression models are the most common, we can add polynomials of any degree. The following figure plots the quadratic and cubic models:



Now, let's try an even higher-order polynomial. The plot in the following figure shows a regression curve created by a ninth-degree polynomial:



The ninth-degree polynomial regression model fits the training data almost exactly! The model's r-squared score, however, is -0.09. We created an extremely complex model that fits the training data exactly, but fails to approximate the real relationship. This problem is called **over-fitting**. The model should induce a general rule to map inputs to outputs; instead, it has memorized the inputs and outputs from the training data. As a result, the

model performs poorly on test data. It predicts that a 16 inch pizza should cost less than \$10, and an 18 inch pizza should cost more than \$30. This model exactly fits the training data, but fails to learn the real relationship between size and price.

Regularization

Regularization is a collection of techniques that can be used to prevent over-fitting. Regularization adds information to a problem, often in the form of a penalty against complexity, to a problem. Occam's razor states that a hypothesis with the fewest assumptions is the best. Accordingly, regularization attempts to find the simplest model that explains the data.

scikit-learn provides several regularized linear regression models. **Ridge regression**, also known as **Tikhonov regularization**, penalizes model parameters that become too large. Ridge regression modifies the residual sum of the squares cost function by adding the L2 norm of the coefficients, as follows:

$$RSS_{\text{ridge}} = \sum_{i=1}^n (y_i - x_i^T \beta)^2 + \lambda \sum_{j=1}^p \beta_j^2$$

λ is a hyperparameter that controls the strength of the penalty. **Hyperparameters** are parameters of the model that are not learned automatically and must be set manually. As λ increases, the penalty increases, and the

value of the cost function increases. When λ is equal to zero, ridge regression is equal to linear regression.

scikit-learn also provides an implementation of the **Least Absolute Shrinkage and Selection Operator (LASSO)**. LASSO penalizes the coefficients by adding their L1 norm to the cost function, as follows:

$$RSS_{\text{lasso}} = \sum_{i=1}^n (y_i - x_i^T \beta)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

The LASSO produces sparse parameters; most of the coefficients will become zero, and the model will depend on a small subset of the features. In contrast, ridge regression produces models in which most parameters are small but nonzero. When explanatory variables are correlated, the LASSO will shrink the coefficients of one variable toward zero. Ridge regression will shrink them more uniformly. Finally, scikit-learn provides an implementation of **elastic net** regularization, which linearly combines the L1 and L2 penalties used by the LASSO and ridge regression. That is, the LASSO and ridge regression are both special cases of the elastic net method in which the hyperparameter for either the L1 or L2 penalty is equal to zero.

Applying linear regression

We have worked through a toy problem to learn how linear regression models relationships between explanatory and response variables. Now we'll use a real data set and apply linear regression to an important task. Assume that you are at a party, and that you wish to drink the best wine that is available. You could ask your friends for recommendations, but you suspect that they will drink any wine, regardless of its provenance. Fortunately, you have brought pH test strips and other tools to measure various physicochemical properties of wine—it is, after all, a party. We will use machine learning to predict the quality of the wine based on its physicochemical attributes.

The UCI Machine Learning Repository's Wine data set measures eleven physicochemical attributes, including the pH and alcohol content, of 1,599 different red wines. Each wine's quality has been scored by human judges. The scores range from zero to ten; zero is the worst quality and ten is the best quality. The data set can be downloaded from

<https://archive.ics.uci.edu/ml/datasets/Wine>. We will approach this problem as a regression task and regress the wine's quality onto one or more physicochemical

attributes. The response variable in this problem takes only integer values between 0 and 10; we could view these as discrete values and approach the problem as a multiclass classification task. In this chapter, however, we will view the response variable as a continuous value.

Exploring the data

Fixed acidity	Volatile acidity	Citric acidity	Residual sugar	Chlorides	Free sulfur dioxide	Total sulfur dioxide	Density	pH
7.4	0.7	0	1.9	0.076	11	34	0.9978	3.5
7.8	0.88	0	2.6	0.098	25	67	0.9968	3.2
7.8	0.76	0.04	2.3	0.092	15	54	0.997	3.2
11.2	0.28	0.56	1.9	0.075	17	60	0.998	3.1

scikit-learn is intended to be a tool to build machine learning systems; its capabilities to explore data are impoverished compared to those of packages such as SPSS Statistics or the R language. We will use pandas, an open source data analysis library for Python, to generate descriptive statistics from the data; we will use these statistics to inform some of the design decisions of our model. pandas introduces Python to some concepts from R such as the dataframe, a two-dimensional, tabular, and heterogeneous data structure. Using pandas for data analysis is the topic of several books; we will use only a few basic methods in the following examples.

First, we will load the data set and review some basic summary statistics for the variables. The data is provided as a .csv file. Note that the fields are separated by semicolons rather than commas):

```
>>> import pandas as pd  
>>> df = pd.read_csv('winequality-red.csv',  
sep=';')  
>>> df.describe()
```

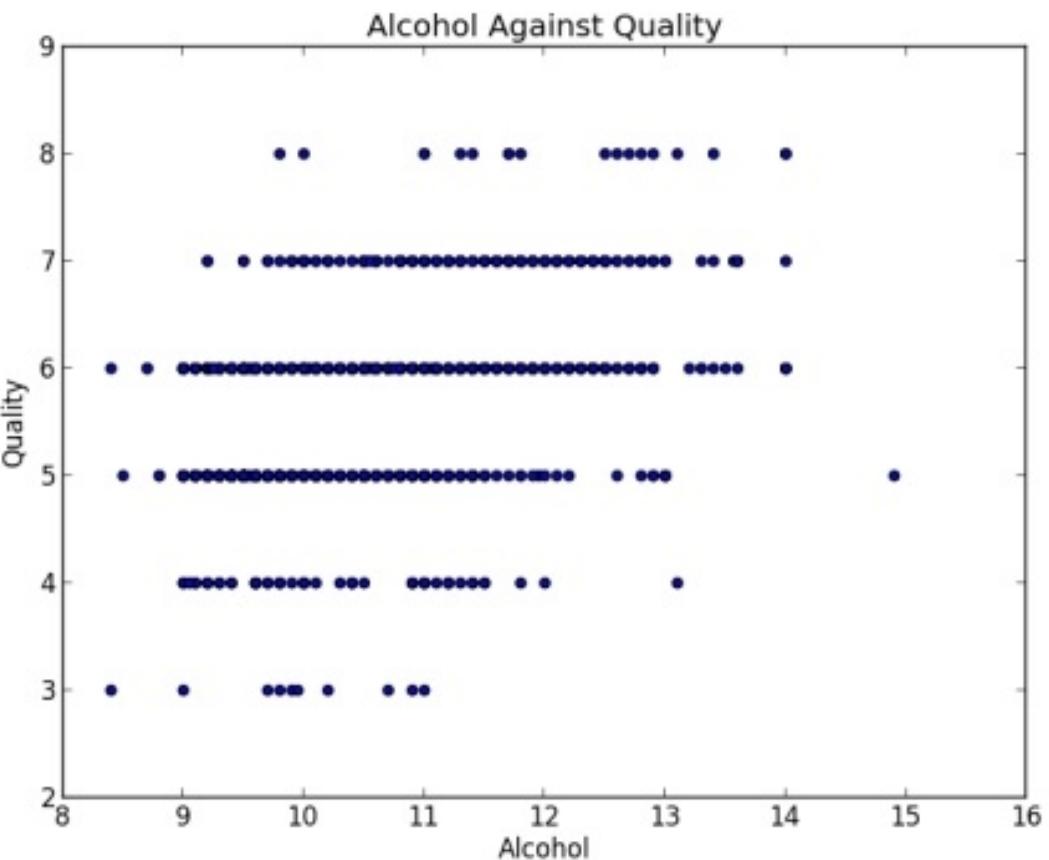
	pH	sulphates	alcohol
quality			
count	1599.000000	1599.000000	1599.000000
1599.000000			
mean	3.311113	0.658149	10.422983
5.636023			
std	0.154386	0.169507	1.065668
0.807569			
min	2.740000	0.330000	8.400000
3.000000			
25%	3.210000	0.550000	9.500000
5.000000			
50%	3.310000	0.620000	10.200000
6.000000			
75%	3.400000	0.730000	11.100000
6.000000			
max	4.010000	2.000000	14.900000
8.000000			

The `pd.read_csv()` function is a convenience utility that loads the .csv file into a dataframe. The Dataframe.`describe()` method calculates summary

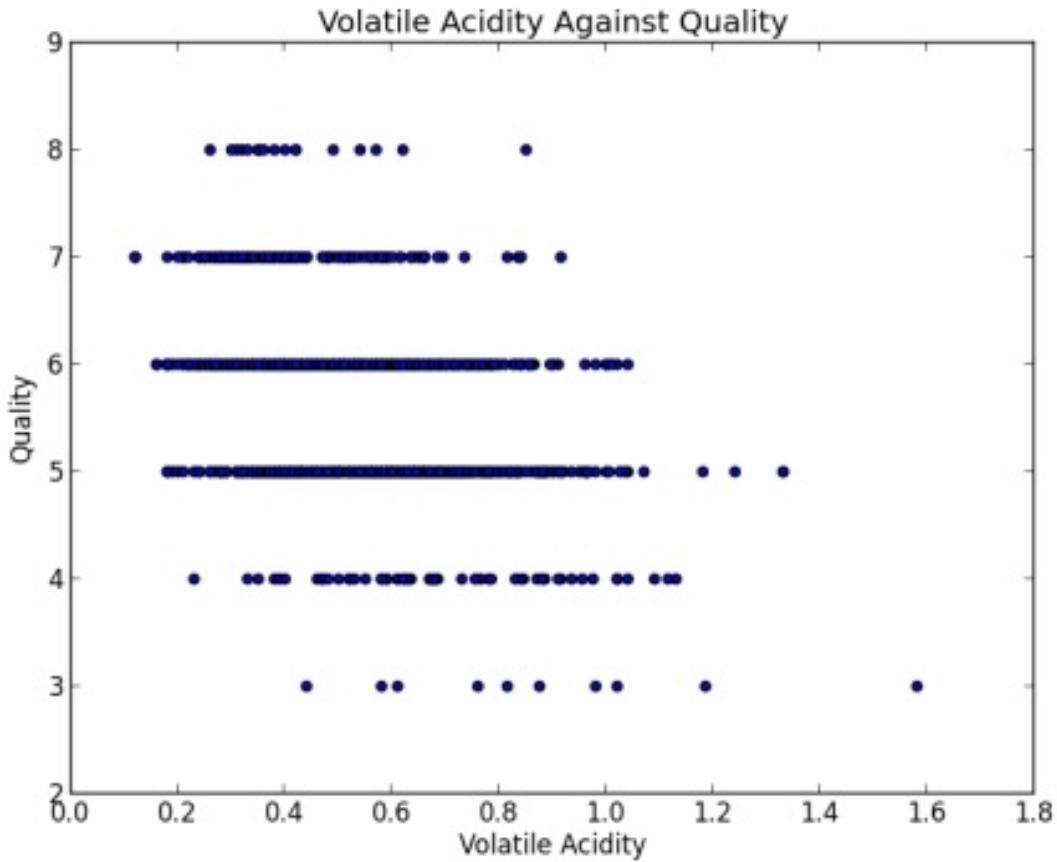
statistics for each column of the dataframe. The preceding code sample shows the summary statistics for only the last four columns of the dataframe. Note the summary for the quality variable; most of the wines scored five or six. Visualizing the data can help indicate if relationships exist between the response variable and the explanatory variables. Let's use `matplotlib` to create some scatter plots. Consider the following code snippet:

```
>>> import matplotlib.pyplot as plt  
>>> plt.scatter(df['alcohol'], df['quality'])  
>>> plt.xlabel('Alcohol')  
>>> plt.ylabel('Quality')  
>>> plt.title('Alcohol Against Quality')  
>>> plt.show()
```

The output of the preceding code snippet is shown in the following figure:



A weak positive relationship between the alcohol content and quality is visible in the scatter plot in the preceding figure; wines that have high alcohol content are often high in quality. The following figure reveals a negative relationship between volatile acidity and quality:



These plots suggest that the response variable depends on multiple explanatory variables; let's model the relationship with multiple linear regression. How can we decide which explanatory variables to include in the model? `Dataframe.corr()` calculates a pairwise correlation matrix. The correlation matrix confirms that the strongest positive correlation is between the alcohol and quality, and that quality is negatively correlated with

volatile acidity, an attribute that can cause wine to taste like vinegar. To summarize, we have hypothesized that good wines have high alcohol content and do not taste like vinegar. This hypothesis seems sensible, though it suggests that wine aficionados may have less sophisticated palates than they claim.

Fitting and evaluating the model

Now we will split the data into training and testing sets, train the regressor, and evaluate its predictions:

```
>>> from sklearn.linear_model import  
LinearRegression  
>>> import pandas as pd  
>>> import matplotlib.pyplot as plt  
>>> from sklearn.cross_validation import  
train_test_split  
  
>>> df = pd.read_csv('wine/winequality-red.csv',  
sep=';')  
>>> X = df[list(df.columns)[:-1]]  
>>> y = df['quality']  
>>> X_train, X_test, y_train, y_test =  
train_test_split(X, y)  
  
>>> regressor = LinearRegression()  
>>> regressor.fit(X_train, y_train)  
>>> y_predictions = regressor.predict(X_test)  
>>> print 'R-squared:', regressor.score(X_test,  
y_test)  
0.345622479617
```

First, we loaded the data using pandas and separated the response variable from the explanatory variables. Next, we used the `train_test_split` function to randomly partition the data into training and test sets. The proportions of the data for both partitions can be specified

using keyword arguments. By default, 25 percent of the data is assigned to the test set. Finally, we trained the model and evaluated it on the test set.

The r-squared score of 0.35 indicates that 35 percent of the variance in the test set is explained by the model. The performance might change if a different 75 percent of the data is partitioned to the training set. We can use cross-validation to produce a better estimate of the estimator's performance. Recall from chapter one that each cross-validation round trains and tests different partitions of the data to reduce variability:

```
>>> import pandas as pd
>>> from sklearn.cross_validation import
cross_val_score
>>> from sklearn.linear_model import
LinearRegression
>>> df = pd.read_csv('data/winequality-red.csv',
sep=';')
>>> X = df[list(df.columns)[:-1]]
>>> y = df['quality']
>>> regressor = LinearRegression()
>>> scores = cross_val_score(regressor, X, y,
cv=5)
>>> print scores.mean(), scores
0.290041628842 [ 0.13200871  0.31858135
0.34955348  0.369145    0.2809196 ]
```

The `cross_val_score` helper function allows us to easily

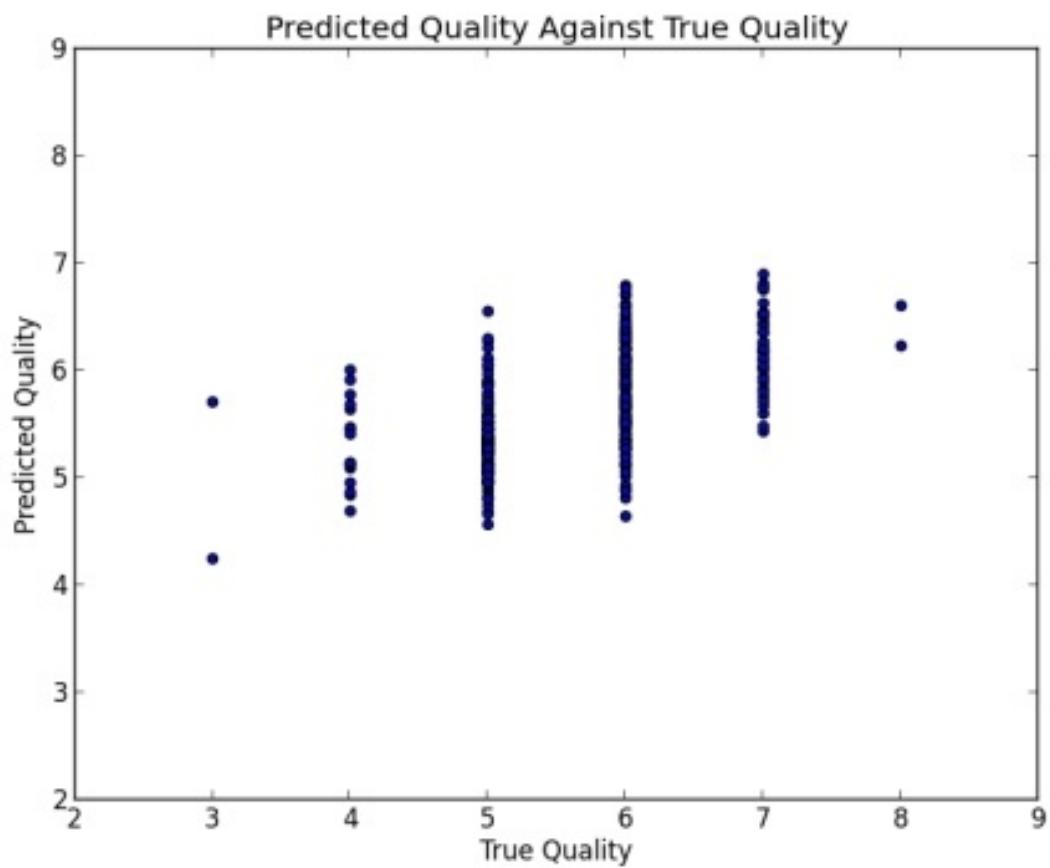
perform cross-validation using the provided data and estimator. We specified a five-fold cross validation using the `cv` keyword argument, that is, each instance will be randomly assigned to one of the five partitions. Each partition will be used to train and test the model.

`cross_val_score` returns the value of the estimator's `score` method for each round. The r-squared scores range from 0.13 to 0.36! The mean of the scores, 0.29, is a better estimate of the estimator's predictive power than the r-squared score produced from a single train / test split.

Let's inspect some of the model's predictions and plot the true quality scores against the predicted scores:

```
Predicted: 4.89907499467 True: 4
Predicted: 5.60701048317 True: 6
Predicted: 5.92154439575 True: 6
Predicted: 5.54405696963 True: 5
Predicted: 6.07869910663 True: 7
Predicted: 6.036656327 True: 6
Predicted: 6.43923020473 True: 7
Predicted: 5.80270760407 True: 6
Predicted: 5.92425033278 True: 5
Predicted: 5.31809822449 True: 6
Predicted: 6.34837585295 True: 6
```

The following figure shows the output of the preceding code:



As expected, few predictions exactly match the true values of the response variable. The model is also better at predicting the qualities of average wines, since most of the training data is for average wines.

Fitting models with gradient descent

In the examples in this chapter, we analytically solved the values of the model's parameters that minimize the cost function with the following equation:

$$\beta = (X^T X)^{-1} X^T Y$$

Recall that X is the matrix of the values of the explanatory variables for each training example. The dot product of $X^T X$ results in a square matrix with dimensions $n \times n$, where n is equal to the number of explanatory variables. The computational complexity of inverting this square matrix is nearly cubic in the number of explanatory variables. While the number of explanatory variables has been small in this chapter's examples, this inversion can be prohibitively costly for problems with tens of thousands of explanatory variables, which we will encounter in the following chapters.

Furthermore, $X^T X$ cannot be inverted if its determinant is equal to zero. In this section, we will discuss another method to efficiently estimate the optimal values of the

model's parameters called **gradient descent**. Note that our definition of a good fit has not changed; we will still use gradient descent to estimate the values of the model's parameters that minimize the value of the cost function.

Gradient descent is sometimes described by the analogy of a blindfolded man who is trying to find his way from somewhere on a mountainside to the lowest point of the valley. He cannot see the topography, so he takes a step in the direction with the steepest decline. He then takes another step, again in the direction with the steepest decline. The sizes of his steps are proportional to the steepness of the terrain at his current position. He takes big steps when the terrain is steep, as he is confident that he is still near the peak and that he will not overshoot the valley's lowest point. The man takes smaller steps as the terrain becomes less steep. If he were to continue taking large steps, he may accidentally step over the valley's lowest point. He would then need to change direction and step toward the lowest point of the valley again. By taking decreasingly large steps, he can avoid stepping back and forth over the valley's lowest point. The blindfolded man continues to walk until he cannot take a step that will decrease his altitude; at this point, he has found the bottom of the valley.

Formally, gradient descent is an optimization algorithm

that can be used to estimate the local minimum of a function. Recall that we are using the residual sum of squares cost function, which is given by the following equation:

$$SS_{res} = \sum_{i=1}^n (y_i - f(x_i))^2$$

We can use gradient descent to find the values of the model's parameters that minimize the value of the cost function. Gradient descent iteratively updates the values of the model's parameters by calculating the partial derivative of the cost function at each step. The calculus required to compute the partial derivative of the cost function is beyond the scope of this book, and is also not required to work with scikit-learn. However, having an intuition for how gradient descent works can help you use it effectively.

It is important to note that gradient descent estimates the local minimum of a function. A three-dimensional plot of the values of a convex cost function for all possible values of the parameters looks like a bowl. The bottom of the bowl is the sole local minimum. Non-convex cost functions can have many local minima, that is, the plots of

the values of their cost functions can have many peaks and valleys. Gradient descent is only guaranteed to find the local minimum; it will find a valley, but will not necessarily find the lowest valley. Fortunately, the residual sum of the squares cost function is convex.

An important hyperparameter of gradient descent is the learning rate, which controls the size of the blindfolded man's steps. If the learning rate is small enough, the cost function will decrease with each iteration until gradient descent has converged on the optimal parameters. As the learning rate decreases, however, the time required for gradient descent to converge increases; the blindfolded man will take longer to reach the valley if he takes small steps than if he takes large steps. If the learning rate is too large, the man may repeatedly overstep the bottom of the valley, that is, gradient descent could oscillate around the optimal values of the parameters.

There are two varieties of gradient descent that are distinguished by the number of training instances that are used to update the model parameters in each training iteration. **Batch gradient descent**, which is sometimes called only gradient descent, uses all of the training instances to update the model parameters in each iteration. **Stochastic Gradient Descent (SGD)**, in contrast, updates the parameters using only a single training instance in

each iteration. The training instance is usually selected randomly. Stochastic gradient descent is often preferred to optimize cost functions when there are hundreds of thousands of training instances or more, as it will converge more quickly than batch gradient descent. Batch gradient descent is a deterministic algorithm, and will produce the same parameter values given the same training set. As a stochastic algorithm, SGD can produce different parameter estimates each time it is run. SGD may not minimize the cost function as well as gradient descent because it uses only single training instances to update the weights. Its approximation is often close enough, particularly for convex cost functions such as residual sum of squares.

Let's use stochastic gradient descent to estimate the parameters of a model with scikit-learn. `SGDRegressor` is an implementation of SGD that can be used even for regression problems with hundreds of thousands or more features. It can be used to optimize different cost functions to fit different linear models; by default, it will optimize the residual sum of squares. In this example, we will predict the prices of houses in the Boston Housing data set from 13 explanatory variables:

```
>>> import numpy as np  
>>> from sklearn.datasets import load_boston  
>>> from sklearn.linear_model import
```

```
SGDRegressor
>>> from sklearn.cross_validation import
cross_val_score
>>> from sklearn.preprocessing import
StandardScaler
>>> from sklearn.cross_validation import
train_test_split
>>> data = load_boston()
>>> X_train, X_test, y_train, y_test =
train_test_split(data.data, data.target)
```

scikit-learn provides a convenience function for loading the data set. First, we split the data into training and testing sets using `train_test_split`:

```
>>> X_scaler = StandardScaler()
>>> y_scaler = StandardScaler()
>>> X_train = X_scaler.fit_transform(X_train)
>>> y_train = y_scaler.fit_transform(y_train)
>>> X_test = X_scaler.transform(X_test)
>>> y_test = y_scaler.transform(y_test)
```

Next, we scaled the features using `StandardScaler`, which we will describe in detail in the next chapter:

```
>>> regressor =
SGDRegressor(loss='squared_loss')
>>> scores = cross_val_score(regressor, X_train,
y_train, cv=5)
>>> print 'Cross validation r-squared scores:', scores
>>> print 'Average cross validation r-squared'
```

```
score:', np.mean(scores)
>>> regressor.fit_transform(X_train, y_train)
>>> print 'Test set r-squared score',
regressor.score(X_test, y_test)
```

Finally, we trained the estimator, and evaluated it using cross validation and the test set. The following is the output of the script:

```
Cross validation r-squared scores: [ 0.73428974
0.80517755  0.58608421  0.83274059  0.69279604]
Average cross validation r-squared score:
0.730217627242
Test set r-squared score 0.653188093125
```

Summary

In this chapter we discussed three cases of linear regression. We worked through an example of simple linear regression, which models the relationship between a single explanatory variable and a response variable using a line. We then discussed multiple linear regression, which generalizes simple linear regression to model the relationship between multiple explanatory variables and a response variable. Finally, we described polynomial regression, a special case of multiple linear regression that models non-linear relationships between explanatory variables and a response variable. These three models can be viewed as special cases of the generalized linear model, a framework for model linear relationships, which we will discuss in more detail in [Chapter 4, From Linear Regression to Logistic Regression.](#)

We assessed the fitness of models using the residual sum of squares cost function and discussed two methods to learn the values of a model's parameters that minimize the cost function. First, we solved the values of the model's parameters analytically. We then discussed gradient descent, a method that can efficiently estimate the optimal values of the model's parameters even when the model has a large number of features. The features in this

chapter's examples were simple measurements of their explanatory variables; it was easy to use them in our models. In the next chapter, you will learn to create features for different types of explanatory variables, including categorical variables, text, and images.

Chapter 3. Feature Extraction and Preprocessing

The examples discussed in the previous chapter used simple numeric explanatory variables, such as the diameter of a pizza. Many machine learning problems require learning from observations of categorical variables, text, or images. In this chapter, you will learn basic techniques for preprocessing data and creating feature representations of these observations. These techniques can be used with the regression models discussed in [Chapter 2, Linear Regression](#), as well as the models we will discuss in subsequent chapters.

Extracting features from categorical variables

Many machine learning problems have **categorical**, or **nominal**, rather than continuous features. For example, an application that predicts a job's salary based on its description might use categorical features such as the job's location. Categorical variables are commonly encoded using **one-of-K** or **one-hot** encoding, in which the explanatory variable is encoded using one binary feature for each of the variable's possible values.

For example, let's assume that our model has a `city` explanatory variable that can take one of three values: New York, San Francisco, or Chapel Hill. One-hot encoding represents this explanatory variable using one binary feature for each of the three possible cities.

In scikit-learn, the `DictVectorizer` class can be used to one-hot encode categorical features:

```
>>> from sklearn.feature_extraction import  
DictVectorizer  
>>> onehot_encoder = DictVectorizer()  
>>> instances = [  
>>>     {'city': 'New York',
```

```
>>>      {'city': 'San Francisco'},  
>>>      {'city': 'Chapel Hill'}>>> ]  
>>> print  
onehot_encoder.fit_transform(instances).toarray()  
)  
[[ 0.  1.  0.] [ 0.  0.  1.][ 1.  0.  0.]]
```

Note that resulting features will not necessarily be ordered in the feature vector as they were encountered. In the first training example, the `city` feature's value is `New York`. The second element in the feature vectors corresponds to the `New York` value and is set to 1 for the first instance. It may seem intuitive to represent the values of a categorical explanatory variable with a single integer feature, but this would encode artificial information. For example, the feature vectors for the previous example would have only one dimension. `New York` could be represented by 0, `San Francisco` by 1, and `Chapel Hill` by 2. This representation would encode an order for the values of the variable that does not exist in the real world; there is no natural order of cities.

Extracting features from text

Many machine learning problems use text as an explanatory variable. Text must be transformed to a different representation that encodes as much of its meaning as possible in a feature vector. In the following sections we will review variations of the most common representation of text that is used in machine learning: the bag-of-words model.

The bag-of-words representation

The most common representation of text is the **bag-of-words** model. This representation uses a multiset, or bag, that encodes the words that appear in a text; the bag-of-words does not encode any of the text's syntax, ignores the order of words, and disregards all grammar. Bag-of-words can be thought of as an extension to one-hot encoding. It creates one feature for each word of interest in the text. The bag-of-words model is motivated by the intuition that documents containing similar words often have similar meanings. The bag-of-words model can be used effectively for document classification and retrieval despite the limited information that it encodes.

A collection of documents is called a **corpus**. Let's use a corpus with the following two documents to examine the bag-of-words model:

```
corpus = [  
    'UNC played Duke in basketball',  
    'Duke lost the basketball game'  
]
```

This corpus contains eight unique words: UNC, played, Duke, in, basketball, lost, the, and game. The corpus's unique words comprise its **vocabulary**. The bag-of-words

model uses a feature vector with an element for each of the words in the corpus's vocabulary to represent each document. Our corpus has eight unique words, so each document will be represented by a vector with eight elements. The number of elements that comprise a feature vector is called the vector's **dimension**. A **dictionary** maps the vocabulary to indices in the feature vector.

In the most basic bag-of-words representation, each element in the feature vector is a binary value that represents whether or not the corresponding word appeared in the document. For example, the first word in the first document is `UNC`. The first word in the dictionary is `UNC`, so the first element in the vector is equal to one. The last word in the dictionary is `game`. The first document does not contain the word `game`, so the eighth element in its vector is set to 0. The `CountVectorizer` class can produce a bag-of-words representation from a string or file. By default, `CountVectorizer` converts the characters in the documents to lowercase, and **tokenizes** the documents. Tokenization is the process of splitting a string into **tokens**, or meaningful sequences of characters. Tokens frequently are words, but they may also be shorter sequences including punctuation characters and affixes. The `CountVectorizer` class tokenizes using a regular expression that splits strings on whitespace and extracts

sequences of characters that are two or more characters in length.

The documents in our corpus are represented by the following feature vectors:

```
>>> from sklearn.feature_extraction.text import  
CountVectorizer  
>>> corpus = [  
>>>     'UNC played Duke in basketball',  
>>>     'Duke lost the basketball game'  
>>> ]  
>>> vectorizer = CountVectorizer()  
>>> print  
vectorizer.fit_transform(corpus).todense()  
>>> print vectorizer.vocabulary_  
[[1 1 0 1 0 1 0 1]  
 [1 1 1 0 1 0 1 0]]  
{u'duke': 1, u'basketball': 0, u'lost': 4,  
 u'played': 5, u'game': 2, u'unc': 7, u'in': 3,  
 u'the': 6}
```

Now let's add a third document to our corpus:

```
corpus = [  
    'UNC played Duke in basketball',  
    'Duke lost the basketball game',  
    'I ate a sandwich'  
]
```

Our corpus's dictionary now contains the following ten unique words. Note that `I` and `a` were not extracted as

they do not match the default regular expression that CountVectorizer uses to tokenize strings:

```
{u'duke': 2, u'basketball': 1, u'lost': 5,  
u'played': 6, u'in': 4, u'game': 3, u'sandwich':  
7, u'unc': 9, u'ate': 0, u'the': 8}
```

Now, our feature vectors are as follows:

```
UNC played Duke in basketball = [[0 1 1 0 1 0 1  
0 0 1]]  
Duke lost the basketball game = [[0 1 1 1 0 1 0  
0 1 0]]  
I ate a sandwich = [[1 0 0 0 0 0 0 1 0 0]]
```

The meanings of the first two documents are more similar to each other than they are to the third document, and their corresponding feature vectors are more similar to each other than they are to the third document's feature vector when using a metric such as **Euclidean distance**. The Euclidean distance between two vectors is equal to the **Euclidean norm**, or L2 norm, of the difference between the two vectors:

$$d = \|x_0 - x_1\|$$

Recall that the Euclidean norm of a vector is equal to the

vector's magnitude, which is given by the following equation:

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$$

scikit-learn's `euclidean_distances` function can be used to calculate the distance between two or more vectors, and it confirms that the most semantically similar documents are also the closest to each other in space. In the following example, we will use the `euclidean_distances` function to compare the feature vectors for our documents:

```
>>> from sklearn.metrics.pairwise import  
euclidean_distances  
>>> counts = [  
>>>     [0, 1, 1, 0, 0, 1, 0, 1],  
>>>     [0, 1, 1, 1, 1, 0, 0, 0],  
>>>     [1, 0, 0, 0, 0, 0, 1, 0]  
>>> ]  
>>> print 'Distance between 1st and 2nd  
documents:', euclidean_distances(counts[0],  
counts[1])  
>>> print 'Distance between 1st and 3rd  
documents:', euclidean_distances(counts[0],  
counts[2])  
>>> print 'Distance between 2nd and 3rd  
documents:', euclidean_distances(counts[1],  
counts[2])
```

```
Distance between 1st and 2nd documents: [[ 2. ]]  
Distance between 1st and 3rd documents: [[  
2.44948974] ]  
Distance between 2nd and 3rd documents: [[  
2.44948974] ]
```

Now let's assume that we are using a corpus of news articles instead of our toy corpus. Our dictionary may now have hundreds of thousands of unique words instead of only twelve. The feature vectors representing the articles will each have hundreds of thousands of elements, and many of the elements will be zero. Most sports articles will not have any of the words particular to finance articles and most culture articles will not have any of the words particular to articles about finance. High-dimensional feature vectors that have many zero-valued elements are called **sparse vectors**.

Using high-dimensional data creates several problems for all machine learning tasks, including those that do not involve text. The first problem is that high-dimensional vectors require more memory than smaller vectors. NumPy provides some data types that mitigate this problem by efficiently representing only the nonzero elements of sparse vectors.

The second problem is known as the **curse of dimensionality**, or the **Hughes effect**. As the feature

space's dimensionality increases, more training data is required to ensure that there are enough training instances with each combination of the feature's values. If there are insufficient training instances for a feature, the algorithm may overfit noise in the training data and fail to generalize. In the following sections, we will review several strategies to reduce the dimensionality of text features. In [Chapter 7](#), *Dimensionality Reduction with PCA*, we will review techniques for numerical dimensionality reduction.

Stop-word filtering

A basic strategy to reduce the dimensions of the feature space is to convert all of the text to lowercase. This is motivated by the insight that the letter case does not contribute to the meanings of most words; sandwich and Sandwich have the same meaning in most contexts.

Capitalization may indicate that a word is at the beginning of a sentence, but the bag-of-words model has already discarded all information from word order and grammar.

A second strategy is to remove words that are common to most of the documents in the corpus. These words, called **stop words**, include determiners such as the, a, and an; auxiliary verbs such as do, be, and will; and prepositions such as on, around, and beneath. Stop words are often functional words that contribute to the document's meaning through grammar rather than their denotations.

The CountVectorizer class can filter stop words provided as the stop_words keyword argument and also includes a basic English stop list. Let's recreate the feature vectors for our documents using stop filtering:

```
>>> from sklearn.feature_extraction.text import  
CountVectorizer  
>>> corpus = [  
>>>     'UNC played Duke in basketball',
```

```
>>>      'Duke lost the basketball game',
>>>      'I ate a sandwich'
>>> ]
>>> vectorizer =
CountVectorizer(stop_words='english')
>>> print
vectorizer.fit_transform(corpus).todense()
>>> print vectorizer.vocabulary_
[[0 1 1 0 0 1 0 1]
 [0 1 1 1 1 0 0 0]
 [1 0 0 0 0 0 1 0]]
{u'duke': 2, u'basketball': 1, u'lost': 4,
u'played': 5, u'game': 3, u'sandwich': 6,
u'unc': 7, u'ate': 0}
```

The feature vectors have now fewer dimensions, and the first two document vectors are still more similar to each other than they are to the third document.

Stemming and lemmatization

While stop filtering is an easy strategy for dimensionality reduction, most stop lists contain only a few hundred words. A large corpus may still have hundreds of thousands of unique words after filtering. Two similar strategies for further reducing dimensionality are called **stemming** and **lemmatization**.

A high-dimensional document vector may separately encode several derived or inflected forms of the same word. For example, `jumping` and `jumps` are both forms of the word `jump`; a document vector in a corpus of long-jumping articles may encode each inflected form with a separate element in the feature vector. Stemming and lemmatization are two strategies to condense inflected and derived forms of a word into a single feature.

Let's consider another toy corpus with two documents:

```
>>> from sklearn.feature_extraction.text import  
CountVectorizer  
>>> corpus = [  
>>>     'He ate the sandwiches',  
>>>     'Every sandwich was eaten by him'  
>>> ]  
>>> vectorizer = CountVectorizer(binary=True,  
stop_words='english')  
>>> print
```

```
vectorizer.fit_transform(corpus).todense()
>>> print vectorizer.vocabulary_
[[1 0 0 1]
 [0 1 1 0]]
{u'sandwich': 2, u'ate': 0, u'sandwiches': 3,
 u'eaten': 1}
```

The documents have similar meanings, but their feature vectors have no elements in common. Both documents contain a conjugation of `ate` and an inflected form of `sandwich`. Ideally, these similarities should be reflected in the feature vectors. Lemmatization is the process of determining the **lemma**, or the morphological root, of an inflected word based on its context. Lemmas are the base forms of words that are used to key the word in a dictionary. **Stemming** has a similar goal to lemmatization, but it does not attempt to produce the morphological roots of words. Instead, stemming removes all patterns of characters that appear to be affixes, resulting in a token that is not necessarily a valid word. Lemmatization frequently requires a lexical resource, like WordNet, and the word's part of speech. Stemming algorithms frequently use rules instead of lexical resources to produce stems and can operate on any token, even without its context.

Let's consider lemmatization of the word `gathering` in two documents:

```
corpus = [
    'I am gathering ingredients for the
sandwich.',
    'There were many wizards at the gathering.'
]
```

In the first sentence `gathering` is a verb, and its lemma is `gather`. In the second sentence `gathering` is a noun, and its lemma is `gathering`. We will use the **Natural Language Tool Kit (NLTK)** to stem and lemmatize the corpus. NLTK can be installed using the instructions at <http://www.nltk.org/install.html>. After installation, execute the following code:

```
>>> import nltk
>>> nltk.download()
```

Then follow the instructions to download the corpora for NLTK.

Using the parts of speech of `gathering`, NLTK's `WordNetLemmatizer` correctly lemmatizes the words in both documents as shown in the following example:

```
>>> from nltk.stem.wordnet import
WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> print lemmatizer.lemmatize('gathering', 'v')
>>> print lemmatizer.lemmatize('gathering', 'n')
gather
```

gathering

Let's compare lemmatization with stemming. The Porter stemmer cannot consider the inflected form's part of speech and returns `gather` for both documents:

```
>>> from nltk.stem import PorterStemmer  
>>> stemmer = PorterStemmer()  
>>> print stemmer.stem('gathering')  
gather
```

Now let's lemmatize our toy corpus:

```
>>> from nltk import word_tokenize  
>>> from nltk.stem import PorterStemmer  
>>> from nltk.stem.wordnet import  
WordNetLemmatizer  
>>> from nltk import pos_tag  
>>> wordnet_tags = ['n', 'v']  
>>> corpus = [  
>>>     'He ate the sandwiches',  
>>>     'Every sandwich was eaten by him'  
>>> ]  
>>> stemmer = PorterStemmer()  
>>> print 'Stemmed:', [[stemmer.stem(token) for  
token in word_tokenize(document)] for document  
in corpus]  
>>> def lemmatize(token, tag):  
>>>     if tag[0].lower() in ['n', 'v']:  
>>>         return lemmatizer.lemmatize(token,  
tag[0].lower())  
>>>     return token  
>>> lemmatizer = WordNetLemmatizer()
```

```
>>> tagged_corpus =  
[pos_tag(word_tokenize(document)) for document  
in corpus]  
>>> print 'Lemmatized:', [[lemmatize(token, tag)  
for token, tag in document] for document in  
tagged_corpus]  
Stemmed: [['He', 'ate', 'the', 'sandwich'],  
['Everi', 'sandwich', 'wa', 'eaten', 'by',  
'him']]  
Lemmatized: [['He', 'eat', 'the', 'sandwich'],  
['Every', 'sandwich', 'be', 'eat', 'by', 'him']]
```

Through stemming and lemmatization, we reduced the dimensionality of our feature space. We produced feature representations that more effectively encode the meanings of the documents despite the fact that the words in the corpus's dictionary are inflected differently in the sentences.

Extending bag-of-words with TF-IDF weights

In the previous section we used the bag-of-words representation to create feature vectors that encode whether or not a word from the corpus's dictionary appears in a document. These feature vectors do not encode grammar, word order, or the frequencies of words. It is intuitive that the frequency with which a word appears in a document could indicate the extent to which a document pertains to that word. A long document that contains one occurrence of a word may discuss an entirely different topic than a document that contains many occurrences of the same word. In this section, we will create feature vectors that encode the frequencies of words, and discuss strategies to mitigate two problems caused by encoding term frequencies.

Instead of using a binary value for each element in the feature vector, we will now use an integer that represents the number of times that the words appeared in the document.

We will use the following corpus. With stop word filtering, the corpus is represented by the following feature vector:

```
>>> from sklearn.feature_extraction.text import  
CountVectorizer  
>>> corpus = ['The dog ate a sandwich, the  
wizard transfigured a sandwich, and I ate a  
sandwich']  
>>> vectorizer =  
CountVectorizer(stop_words='english')  
>>> print  
vectorizer.fit_transform(corpus).todense()  
[[2 1 3 1 1]]  
{u'sandwich': 2, u'wizard': 4, u'dog': 1,  
u'transfigured': 3, u'ate': 0}
```

The element for `dog` is now set to 1 and the element for `sandwich` is set to 2 to indicate that the corresponding words occurred once and twice, respectively. Note that the `binary` keyword argument of `CountVectorizer` is omitted; its default value is `False`, which causes it to return raw term frequencies rather than binary frequencies. Encoding the terms' raw frequencies in the feature vector provides additional information about the meanings of the documents, but assumes that all of the documents are of similar lengths.

Many words might appear with the same frequency in two documents, but the documents could still be dissimilar if one document is many times larger than the other. scikit-learn's `TfidfTransformer` object can mitigate this problem by transforming a matrix of term frequency vectors into a

matrix of normalized term frequency weights. By default, `TfidfTransformer` smoothes the raw counts and applies L2 normalization. The smoothed, normalized term frequencies are given by the following equation:

$$tf(t, d) = \frac{f(t, d) + 1}{\|x\|}$$

$f(t, d)$ is the frequency of term t in document d and $\|x\|$ is the L2 norm of the count vector. In addition to normalizing raw term counts, we can improve our feature vectors by calculating **logarithmically scaled term frequencies**, which scale the counts to a more limited range, or **augmented term frequencies**, which further mitigates the bias for longer documents. Logarithmically scaled term frequencies are given by the following equation:

$$tf(t, d) = \log(f(t, d) + 1)$$

The `TfidfTransformer` object calculates logarithmically scaled term frequencies when its `sublinear_tf` keyword

argument is set to `True`. Augmented frequencies are given by the following equation:

$$tf(t,d) = 0.5 + \frac{0.5 * f(t,d)}{\max\{f(w,d) : w \in d\}}$$

$\max\{f(w,d) : w \in d\}$ is the greatest frequency of all of the words in document d . scikit-learn 0.15.2 does not implement augmented term frequencies, but the output of `CountVectorizer` can be easily transformed.

Normalization, logarithmically scaled term frequencies, and augmented term frequencies can represent the frequencies of terms in a document while mitigating the effects of different document sizes. However, another problem remains with these representations. The feature vectors contain large weights for terms that occur frequently in a document, even if those terms occur frequently in most documents in the corpus. These terms do not help to represent the meaning of a particular document relative to the rest of the corpus. For example, most of the documents in a corpus of articles about Duke's basketball team could include the words `basketball`, `Coach K`, and `flop`. These words can be thought of as

corpus-specific stop words and may not be useful to calculate the similarity of documents. The **inverse document frequency (IDF)** is a measure of how rare or common a word is in a corpus. The inverse document frequency is given by the following equation:

$$idf(t, D) = \log \frac{N}{1 + |d \in D : t \in d|}$$

Here, N is the total number of documents in the corpus and $d \in D : t \in d$ is the number of documents in the corpus that contain the term t . A term's **TF-IDF** value is the product of its term frequency and inverse document frequency. `TfidfTransformer` returns TF-IDF's weight when its `use_idf` keyword argument is set to its default value, `True`. Since TF-IDF weighted feature vectors are commonly used to represent text, scikit-learn provides a `TfidfVectorizer` class that wraps `CountVectorizer` and `TfidfTransformer`. Let's use `TfidfVectorizer` to create TF-IDF weighted feature vectors for our corpus:

```
>>> from sklearn.feature_extraction.text import  
TfidfVectorizer  
>>> corpus = [  
>>>     'The dog ate a sandwich and I ate a  
sandwich',
```

```
>>>      'The wizard transfigured a sandwich'  
>>> ]  
>>> vectorizer =  
TfidfVectorizer(stop_words='english')  
>>> print  
vectorizer.fit_transform(corpus).todense()  
[[ 0.75458397  0.37729199  0.53689271  0.  
 0.          ]  
 [ 0.           0.           0.44943642  0.6316672  
 0.6316672 ]]
```

By comparing the TF-IDF weights to the raw term frequencies, we can see that words that are common to many of the documents in the corpus, such as sandwich, have been penalized.

Space-efficient feature vectorizing with the hashing trick

In this chapter's previous examples, a dictionary containing all of the corpus's unique tokens is used to map a document's tokens to the elements of a feature vector. Creating this dictionary has two drawbacks. First, two passes are required over the corpus: the first pass is used to create the dictionary and the second pass is used to create feature vectors for the documents. Second, the dictionary must be stored in memory, which could be prohibitive for large corpora. It is possible to avoid creating this dictionary through applying a hash function to the token to determine its index in the feature vector directly. This shortcut is called the **hashing trick**. The following example uses `HashingVectorizer` to demonstrate the hashing trick:

```
>>> from sklearn.feature_extraction.text import  
HashingVectorizer  
>>> corpus = ['the', 'ate', 'bacon', 'cat']  
>>> vectorizer = HashingVectorizer(n_features=6)  
>>> print vectorizer.transform(corpus).todense()  
[[ -1.  0.  0.  0.  0.]  
 [ 0.  0.  0.  1.  0.  0.]  
 [ 0.  0.  0.  0. -1.  0.]]
```

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

The hashing trick is stateless. It can be used to create feature vectors in both parallel and online, or streaming, applications because it does not require an initial pass over the corpus.. Note that `n_features` is an optional keyword argument. Its default value, 2^{20} , is adequate for most problems; it is set to 6 here so that the matrix will be small enough to print and still display all of the nonzero features. Also, note that some of the term frequencies are negative. Since hash collisions are possible,

`HashingVectorizer` uses a signed hash function. The value of a feature takes the same sign as its token's hash; if the term `cats` appears twice in a document and is hashed to -3, the fourth element of the document's feature vector will be decremented by two. If the term `dogs` also appears twice and is hashed to 3, the fourth element of the feature vector will be incremented by two. Using a signed hash function creates the possibility that errors from hash collisions will cancel each other out rather than accumulate; a loss of information is preferable to a loss of information and the addition of spurious information.

Another disadvantage of the hashing trick is that the resulting model is more difficult to inspect, as the hashing function cannot recall what input token is mapped to each element of the feature vector.

Extracting features from images

Computer vision is the study and design of computational artifacts that process and understand images. These artifacts sometimes employ machine learning. An overview of computer vision is far beyond the scope of this book, but in this section we will review some basic techniques used in computer vision to represent images in machine learning problems.

Extracting features from pixel intensities

A digital image is usually a raster, or pixmap, that maps colors to coordinates on a grid. An image can be viewed as a matrix in which each element represents a color. A basic feature representation for an image can be constructed by reshaping the matrix into a vector by concatenating its rows together. **Optical character recognition (OCR)** is a canonical machine learning problem. Let's use this technique to create basic feature representations that could be used in an OCR application for recognizing hand-written digits in character-delimited forms.

The `digits` dataset included with scikit-learn contains grayscale images of more than 1,700 hand-written digits between zero and nine. Each image has eight pixels on a side. Each pixel is represented by an intensity value between zero and 16; white is the most intense and is indicated by zero, and black is the least intense and is indicated by 16. The following figure is an image of a hand-written digit taken from the dataset:



Let's create a feature vector for the image by reshaping its 8×8 matrix into a **64-dimensional** vector:

```
>>> from sklearn import datasets  
>>> digits = datasets.load_digits()  
>>> print 'Digit:', digits.target[0]  
>>> print digits.images[0]  
>>> print 'Feature vector:\n',  
      digits.images[0].reshape(-1, 64)  
Digit: 0
```

```
[ [ 0.  0.  5.  13.  9.  1.  0.  0.]  
[ 0.  0.  13.  15.  10.  15.  5.  0.]  
[ 0.  3.  15.  2.  0.  11.  8.  0.]  
[ 0.  4.  12.  0.  0.  8.  8.  0.]  
[ 0.  5.  8.  0.  0.  9.  8.  0.]  
[ 0.  4.  11.  0.  1.  12.  7.  0.]  
[ 0.  2.  14.  5.  10.  12.  0.  0.]  
[ 0.  0.  6.  13.  10.  0.  0.  0.] ]
```

Feature vector:

```
[ [ 0.  0.  5.  13.  9.  1.  0.  0.  0.  
0.  13.  15.  10.  15.  
5.  0.  0.  3.  15.  2.  0.  11.  8.  
0.  0.  4.  12.  0.  
0.  8.  8.  0.  0.  5.  8.  0.  0.  
9.  8.  0.  0.  4.  
11. 0.  1.  12.  7.  0.  0.  2.  14.  
5.  10.  12.  0.  0.  
0.  0.  6.  13.  10.  0.  0.  0.] ]
```

This representation can be effective for some basic tasks, like recognizing printed characters. However, recording the intensity of every pixel in the image produces prohibitively large feature vectors. A tiny 100 x 100 grayscale image would require a 10,000-dimensional vector, and a 1920 x 1080 grayscale image would require a 2,073,600-dimensional vector. Unlike the TF-IDF feature vectors we created, in most problems these vectors are not sparse. Space-complexity is not the only disadvantage of this representation; learning from the intensities of pixels at particular locations results in

models that are sensitive to changes in the scale, rotation, and translation of images. A model trained on our basic feature representations might not be able to recognize the same zero if it were shifted a few pixels in any direction, enlarged, or rotated a few degrees. Furthermore, learning from pixel intensities is itself problematic, as the model can become sensitive to changes in illumination. For these reasons, this representation is ineffective for tasks that involve photographs or other natural images. Modern computer vision applications frequently use either hand-engineered feature extraction methods that are applicable to many different problems, or automatically learn features without supervision problem using techniques such as deep learning. We will focus on the former in the next section.

Extracting points of interest as features

The feature vector we created previously represents every pixel in the image; all of the informative attributes of the image are represented and all of the noisy attributes are represented too. After inspecting the training data, we can see that all of the images have a perimeter of white pixels; these pixels are not useful features. Humans can quickly recognize many objects without observing every attribute of the object. We can recognize a car from the contours of the hood without observing the rear-view mirrors, and we can recognize an image of a human face from a nose or mouth. This intuition is motivation to create representations of only the most informative attributes of an image. These informative attributes, or **points of interest**, are points that are surrounded by rich textures and can be reproduced despite perturbing the image.

Edges and **corners** are two common types of points of interest. An edge is a boundary at which pixel intensity rapidly changes, and a corner is an intersection of two edges. Let's use scikit-image to extract points of interest from the following figure:



The code to extract the points of interest is as follows:

```
>>> import numpy as nps  
>>> from skimage.feature import corner_harris,  
corner_peaks
```

```

>>> from skimage.color import rgb2gray
>>> import matplotlib.pyplot as plt
>>> import skimage.io as io
>>> from skimage.exposure import equalize_hist

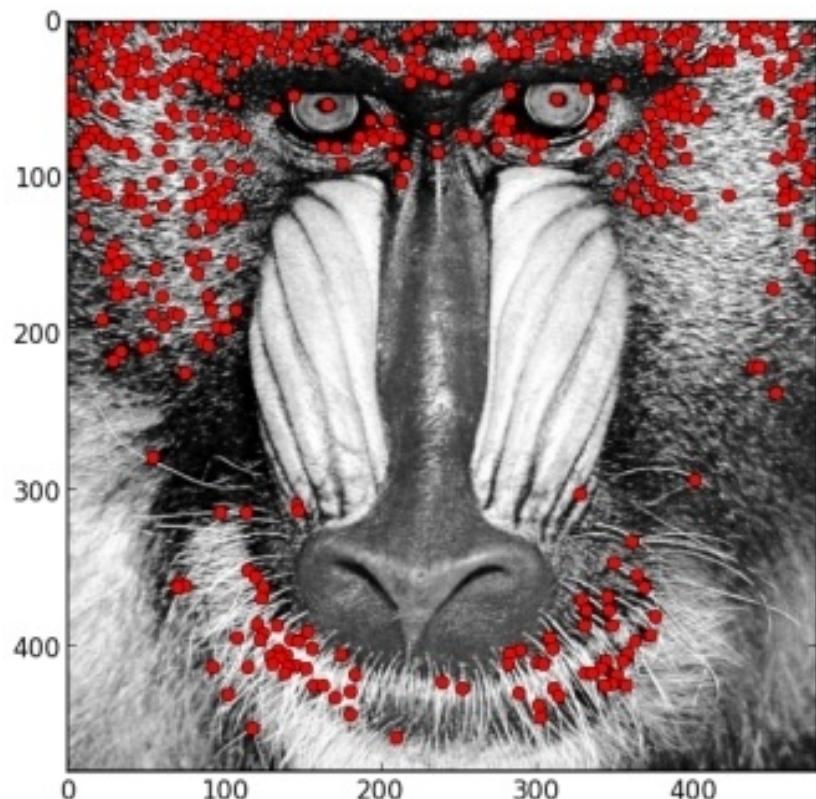
>>> def show_corners(corners, image):
>>>     fig = plt.figure()
>>>     plt.gray()
>>>     plt.imshow(image)
>>>     y_corner, x_corner = zip(*corners)
>>>     plt.plot(x_corner, y_corner, 'or')
>>>     plt.xlim(0, image.shape[1])
>>>     plt.ylim(image.shape[0], 0)
>>>
fig.set_size_inches(np.array(fig.get_size_inches()
()) * 1.5)
>>>     plt.show()

>>> mandrill =
io.imread('/home/gavin/PycharmProjects/mastering
-machine-learning/ch4/img/mandrill.png')
>>> mandrill = equalize_hist(rgb2gray(mandrill))
>>> corners =
corner_peaks(corner_harris(mandrill),
min_distance=2)
>>> show_corners(corners, mandrill)

```

The following figure plots the extracted points of interest. Of the image's 230400 pixels, 466 were extracted as points of interest. This representation is much more compact; ideally, there is enough variation proximal to the points of interest to reproduce them despite changes in the

image's illumination.



SIFT and SURF

Scale-Invariant Feature Transform (SIFT) is a method for extracting features from an image that is less sensitive to the scale, rotation, and illumination of the image than the extraction methods we have previously discussed. Each SIFT feature, or descriptor, is a vector that describes edges and corners in a region of an image. Unlike the points of interest in our previous example, SIFT also captures information about the composition of each point of interest and its surroundings. **Speeded-Up Robust Features (SURF)** is another method of extracting interesting points of an image and creating descriptions that are invariant of the image's scale, orientation, and illumination. SURF can be computed more quickly than SIFT, and it is more effective at recognizing features across images that have been transformed in certain ways.

Explaining how SIFT and SURF extraction are implemented is beyond the scope of this book. However, with an intuition for how they work, we can still effectively use libraries that implement them.

In this example, we will extract SURF from the following image using the `mahotas` library.



Like the extracted points of interest, the extracted SURF are only the first step in creating a feature representation that could be used in a machine learning task. Different

SURF will be extracted for each instance in the training set. In [Chapter 6](#), *Clustering with K-Means*, we will cluster extracted SURF to learn features that can be used by an image classifier. In the following example we will use mahotas to extract SURF descriptors:

```
>>> import mahotas as mh
>>> from mahotas.features import surf

>>> image = mh.imread('zipper.jpg',
as_grey=True)
>>> print 'The first SURF descriptor:\n',
surf.surf(image) [0]
>>> print 'Extracted %s SURF descriptors' % len(surf.surf(image))
The first SURF descriptor:
[ 6.73839947e+02   2.24033945e+03
 3.18074483e+00   2.76324459e+03
 -1.00000000e+00   1.61191475e+00
 4.44035121e-05   3.28041690e-04
  2.44845817e-04   3.86297608e-04
 -1.16723672e-03  -8.81290243e-04
  1.65414959e-03   1.28393061e-03
 -7.45077384e-04   7.77655540e-04
  1.16078772e-03   1.81434398e-03
 1.81736394e-04  -3.13096961e-04
  3.06559785e-04   3.43443699e-04
 2.66200498e-04  -5.79522387e-04
  1.17893036e-03   1.99547411e-03
 -2.25938217e-01  -1.85563853e-01
  2.27973631e-01   1.91510135e-01
 -2.49315698e-01   1.95451021e-01
```

2.59719480e-01	1.98613061e-01
-7.82458546e-04	1.40287015e-03
2.86712113e-03	3.15971628e-03
4.98444730e-04	-6.93986983e-04
1.87531652e-03	2.19041521e-03
1.80681053e-01	-2.70528820e-01
2.32414943e-01	2.72932870e-01
2.65725332e-01	3.28050743e-01
2.98609869e-01	3.41623138e-01
1.58078002e-03	-4.67968721e-04
2.35704122e-03	2.26279888e-03
6.43115065e-06	1.22501486e-04
1.20064616e-04	1.76564805e-04
2.14148537e-03	8.36243899e-05
2.93382280e-03	3.10877776e-03
4.53469215e-03	-3.15254535e-04
6.92437341e-03	3.56880279e-03
-1.95228401e-04	3.73674995e-05
7.02700555e-04	5.45156362e-04]

Extracted 994 SURF descriptors

Data standardization

Many estimators perform better when they are trained on standardized data sets. Standardized data has **zero mean** and **unit variance**. An explanatory variable with zero mean is centered about the origin; its average value is zero. A feature vector has unit variance when the variances of its features are all of the same order of magnitude. For example, assume that a feature vector encodes two explanatory variables. The first values of the first variable range from zero to one. The values of the second explanatory variable range from zero to 100,000. The second feature must be scaled to a range closer to $\{0,1\}$ for the data to have unit variance. If a feature's variance is orders of magnitude greater than the variances of the other features, that feature may dominate the learning algorithm and prevent it from learning from the other variables. Some learning algorithms also converge to the optimal parameter values more slowly when data is not standardized. The value of an explanatory variable can be standardized by subtracting the variable's mean and dividing the difference by the variable's standard deviation. Data can be easily standardized using scikit-learn's `scale` function:

```
>>> from sklearn import preprocessing
```

```
>>> import numpy as np
>>> X = np.array([
>>>     [0., 0., 5., 13., 9., 1.],
>>>     [0., 0., 13., 15., 10., 15.],
>>>     [0., 3., 15., 2., 0., 11.]
>>> ])
>>> print preprocessing.scale(X)
[[ 0.          -0.70710678 -1.38873015
  0.52489066   0.59299945 -1.35873244]
 [ 0.          -0.70710678  0.46291005
  0.87481777   0.81537425  1.01904933]
 [ 0.          1.41421356  0.9258201
 -1.39970842 -1.4083737   0.33968311]]
```

Summary

In this chapter, we discussed feature extraction and developed an understanding about the basic techniques for transforming arbitrary data into feature representations that can be used by machine learning algorithms. First, we created features from categorical explanatory variables using one-hot encoding and scikit-learn's `DictVectorizer`. Then, we discussed the creation of feature vectors for one of the most common types of data used in machine learning problems: text. We worked through several variations of the bag-of-words model, which discards all syntax and encodes only the frequencies of the tokens in a document. We began by creating basic binary term frequencies with `CountVectorizer`. You learned to preprocess text by filtering stop words and stemming tokens, and you also replaced the term counts in our feature vectors with TF-IDF weights that penalize common words and normalize for documents of different lengths. Next, we created feature vectors for images. We began with an optical character recognition problem in which we represented images of hand-written digits with flattened matrices of pixel intensities. This is a computationally costly approach. We improved our representations of images by extracting only their most interesting points as SURF

descriptors.

Finally, you learned to standardize data to ensure that our estimators can learn from all of the explanatory variables and can converge as quickly as possible. We will use these feature extraction techniques in the subsequent chapters' examples. In the next chapter, we will combine the bag-of-words representation with a generalization of multiple linear regressions to classify documents.

Chapter 4. From Linear Regression to Logistic Regression

In [Chapter 2](#), *Linear Regression*, we discussed simple linear regression, multiple linear regression, and polynomial regression. These models are special cases of the generalized linear model, a flexible framework that requires fewer assumptions than ordinary linear regression. In this chapter, we will discuss some of these assumptions as they relate to another special case of the generalized linear model called **logistic regression**.

Unlike the models we discussed previously, logistic regression is used for classification tasks. Recall that the goal in classification tasks is to find a function that maps an observation to its associated class or label. A learning algorithm must use pairs of feature vectors and their corresponding labels to induce the values of the mapping function's parameters that produce the best classifier, as measured by a particular performance metric. In binary classification, the classifier must assign instances to one of the two classes. Examples of binary classification include predicting whether or not a patient has a particular

disease, whether or not an audio sample contains human speech, or whether or not the Duke men's basketball team will lose in the first round of the NCAA tournament. In multiclass classification, the classifier must assign one of several labels to each instance. In multilabel classification, the classifier must assign a subset of the labels to each instance. In this chapter, we will work through several classification problems using logistic regression, discuss performance measures for the classification task, and apply some of the feature extraction techniques you learned in the previous chapter.

Binary classification with logistic regression

Ordinary linear regression assumes that the response variable is normally distributed. The **normal distribution**, also known as the **Gaussian distribution** or **bell curve**, is a function that describes the probability that an observation will have a value between any two real numbers. Normally distributed data is symmetrical. That is, half of the values are greater than the mean and the other half of the values are less than the mean. The mean, median, and mode of normally distributed data are also equal. Many natural phenomena approximately follow normal distributions. For instance, the height of people is normally distributed; most people are of average height, a few are tall, and a few are short.

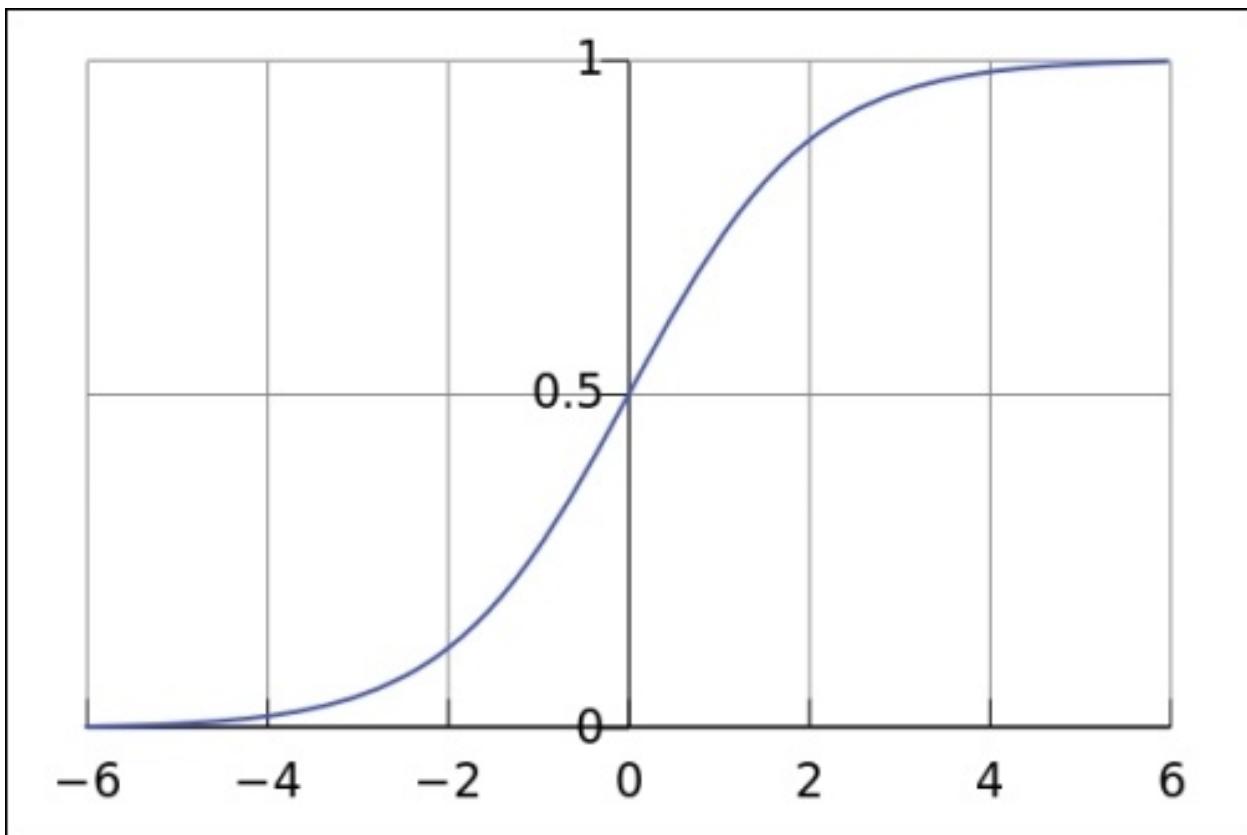
In some problems the response variable is not normally distributed. For instance, a coin toss can result in two outcomes: heads or tails. The **Bernoulli distribution** describes the probability distribution of a random variable that can take the positive case with probability P or the negative case with probability $1-P$. If the response variable represents a probability, it must be constrained to the range $\{0,1\}$. Linear regression assumes that a constant

change in the value of an explanatory variable results in a constant change in the value of the response variable, an assumption that does not hold if the value of the response variable represents a probability. Generalized linear models remove this assumption by relating a linear combination of the explanatory variables to the response variable using a link function. In fact, we already used a link function in [Chapter 2](#), *Linear Regression*; ordinary linear regression is a special case of the generalized linear model that relates a linear combination of the explanatory variables to a normally distributed response variable using the **identity link function**. We can use a different link function to relate a linear combination of the explanatory variables to the response variable that is not normally distributed.

In logistic regression, the response variable describes the probability that the outcome is the positive case. If the response variable is equal to or exceeds a discrimination threshold, the positive class is predicted; otherwise, the negative class is predicted. The response variable is modeled as a function of a linear combination of the explanatory variables using the **logistic function**. Given by the following equation, the logistic function always returns a value between zero and one:

$$F(t) = \frac{1}{1+e^{-t}}$$

The following is a plot of the value of the logistic function for the range $\{-6, 6\}$:



For logistic regression, t is equal to a linear combination of explanatory variables, as follows:

$$F(t) = \frac{1}{1 + e^{-(\beta_0 + \beta_x t)}}$$

The **logit function** is the inverse of the logistic function.
It links $F(x)$ back to a linear combination of the explanatory variables:

$$g(x) = \ln \frac{F(x)}{1 - F(x)} = \beta_0 + \beta_x$$

Now that we have defined the model for logistic regression, let's apply it to a binary classification task.

Spam filtering

Our first problem is a modern version of the canonical binary classification problem: spam classification. In our version, however, we will classify spam and ham SMS messages rather than e-mail. We will extract TF-IDF features from the messages using techniques you learned in [Chapter 3, Feature Extraction and Preprocessing](#), and classify the messages using logistic regression.

We will use the SMS Spam Classification Data Set from the UCI Machine Learning Repository. The dataset can be downloaded from

<http://archive.ics.uci.edu/ml/datasets/SMS+Collection>

First, let's explore the data set and calculate some basic summary statistics using pandas:

```
>>> import pandas as pd  
>>> df = pd.read_csv('data/SMSSpamCollection',  
delimiter='\t', header=None)  
>>> print df.head()  
  
0  
1  
0 ham Go until jurong point, crazy..  
Available only ...  
1 ham Ok lar... Joking  
wif u oni...  
2 spam Free entry in 2 a wkly comp to win FA
```

```
Cup fina...
3    ham  U dun say so early hor... U c already
then say...
4    ham  Nah I don't think he goes to usf, he
lives aro...
[5 rows x 2 columns]
```

```
>>> print 'Number of spam messages:', df[df[0]
== 'spam'][0].count()
>>> print 'Number of ham messages:', df[df[0] ==
'ham'][0].count()
```

```
Number of spam messages: 747
Number of ham messages: 4825
```

A binary label and a text message comprise each row. The data set contains 5,574 instances; 4,827 messages are ham and the remaining 747 messages are spam. The ham messages are labeled with zero, and the spam messages are labeled with one. While the noteworthy, or case, outcome is often assigned the label one and the non-case outcome is often assigned zero, these assignments are arbitrary. Inspecting the data may reveal other attributes that should be captured in the model. The following selection of messages characterizes both of the classes:

Spam: Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive entry question(std txt rate)T&C's apply
08452810075over18's

Spam: WINNER!! As a valued network customer you

have been selected to receivea £900 prize reward! To claim call 09061701461. Claim code KL341. Valid 12 hours only.

Ham: Sorry my roommates took forever, it ok if I come by now?

Ham: Finished class where are you.

Let's make some predictions using scikit-learn's LogisticRegression class:

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import
TfidfVectorizer
>>> from sklearn.linear_model.logistic import
LogisticRegression
>>> from sklearn.cross_validation import
train_test_split, cross_val_score
```

First, we load the .csv file using pandas and split the data set into training and test sets. By default, train_test_split() assigns 75 percent of the samples to the training set and allocates the remaining 25 percent of the samples to the test set:

```
>>> df = pd.read_csv('data/SMSpamCollection',
delimiter='\t', header=None)
>>> X_train_raw, X_test_raw, y_train, y_test =
train_test_split(df[1], df[0])
```

Next, we create a TfidfVectorizer. Recall from [Chapter](#)

[3](#), *Feature Extraction and Preprocessing*, that TfIdfVectorizer combines CountVectorizer and TfIdfTransformer. We fit it with the training messages, and transform both the training and test messages:

```
>>> vectorizer = TfIdfVectorizer()  
>>> X_train =  
vectorizer.fit_transform(X_train_raw)  
>>> X_test = vectorizer.transform(X_test_raw)
```

Finally, we create an instance of LogisticRegression and train our model. Like LinearRegression, LogisticRegression implements the fit() and predict() methods. As a sanity check, we printed a few predictions for manual inspection:

```
>>> classifier = LogisticRegression()  
>>> classifier.fit(X_train, y_train)  
>>> predictions = classifier.predict(X_test)  
>>> for i, prediction in  
enumerate(predictions[:5]):  
>>>     print X_test_raw.values[i],  
'prediction:', prediction
```

The following is the output of the script:

Prediction: ham. Message: If you don't respond imma assume you're still asleep and imma start calling n shit

Prediction: spam. Message: HOT LIVE FANTASIES call now 08707500020 Just 20p per min NTT Ltd,

PO Box 1327 Croydon CR9 5WB 0870 is a national rate call

Prediction: ham. Message: Yup... I havent been there before... You want to go for the yoga? I can call up to book

Prediction: ham. Message: Hi, can i please get a <#> dollar loan from you. I.ll pay you back by mid february. Pls.

Prediction: ham. Message: Where do you need to go to get it?

How well does our classifier perform? The performance metrics we used for linear regression are inappropriate for this task. We are only interested in whether the predicted class was correct, not how far it was from the decision boundary. In the next section, we will discuss some performance metrics that can be used to evaluate binary classifiers.

Binary classification performance metrics

A variety of metrics exist to evaluate the performance of binary classifiers against trusted labels. The most common metrics are **accuracy**, **precision**, **recall**, **F1 measure**, and **ROC AUC score**. All of these measures depend on the concepts of **true positives**, **true negatives**, **false positives**, and **false negatives**. *Positive* and *negative* refer to the classes. *True* and *false* denote whether the predicted class is the same as the true class.

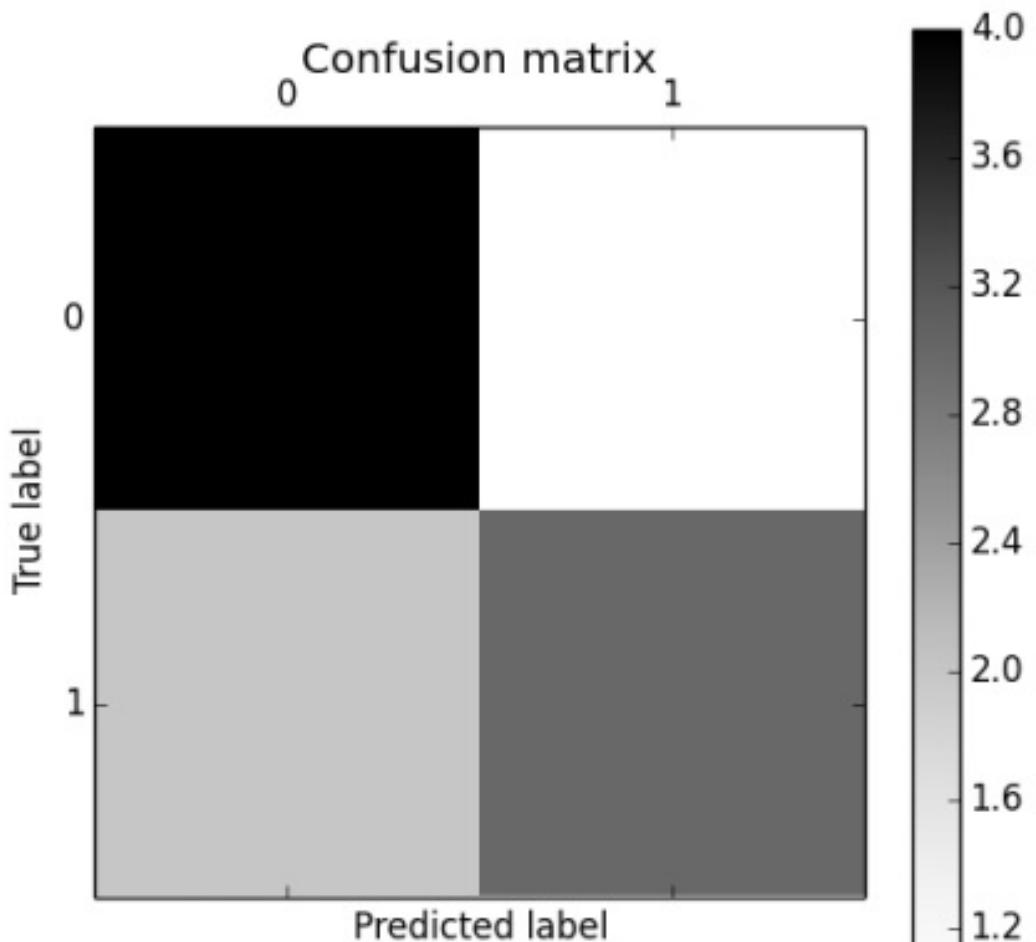
For our SMS spam classifier, a true positive prediction is when the classifier correctly predicts that a message is spam. A true negative prediction is when the classifier correctly predicts that a message is ham. A prediction that a ham message is spam is a false positive prediction, and a spam message incorrectly classified as ham is a false negative prediction. A **confusion matrix**, or **contingency table**, can be used to visualize true and false positives and negatives. The rows of the matrix are the true classes of the instances, and the columns are the predicted classes of the instances:

```
>>> from sklearn.metrics import confusion_matrix  
>>> import matplotlib.pyplot as plt
```

```
>>> y_test = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
>>> y_pred = [0, 1, 0, 0, 0, 0, 1, 1, 1, 1]
>>> confusion_matrix = confusion_matrix(y_test,
y_pred)
>>> print(confusion_matrix)
>>> plt.matshow(confusion_matrix)
>>> plt.title('Confusion matrix')
>>> plt.colorbar()
>>> plt.ylabel('True label')
>>> plt.xlabel('Predicted label')
>>> plt.show()

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

The confusion matrix indicates that there were four true negative predictions, three true positive predictions, two false negative predictions, and one false positive prediction. Confusion matrices become more useful in multi-class problems, in which it can be difficult to determine the most frequent types of errors.



Accuracy

Accuracy measures a fraction of the classifier's predictions that are correct. scikit-learn provides a function to calculate the accuracy of a set of predictions given the correct labels:

```
>>> from sklearn.metrics import accuracy_score  
>>> y_pred, y_true = [0, 1, 1, 0], [1, 1, 1, 1]  
>>> print 'Accuracy:', accuracy_score(y_true,  
y_pred)
```

Accuracy: 0.5

LogisticRegression.score() predicts and scores labels for a test set using accuracy. Let's evaluate our classifier's accuracy:

```
>>> import numpy as np  
>>> import pandas as pd  
>>> from sklearn.feature_extraction.text import  
TfidfVectorizer  
>>> from sklearn.linear_model.logistic import  
LogisticRegression  
>>> from sklearn.cross_validation import  
train_test_split, cross_val_score  
>>> df = pd.read_csv('data/sms.csv')  
>>> X_train_raw, X_test_raw, y_train, y_test =  
train_test_split(df['message'], df['label'])  
>>> vectorizer = TfidfVectorizer()  
>>> X_train =
```

```
vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> scores = cross_val_score(classifier,
X_train, y_train, cv=5)
>>> print np.mean(scores), scores

Accuracy 0.956217208018 [ 0.96057348  0.95334928
0.96411483  0.95454545  0.94850299]
```

Note that your accuracy may differ as the training and test sets are assigned randomly. While accuracy measures the overall correctness of the classifier, it does not distinguish between false positive errors and false negative errors. Some applications may be more sensitive to false negatives than false positives, or vice versa. Furthermore, accuracy is not an informative metric if the proportions of the classes are skewed in the population. For example, a classifier that predicts whether or not credit card transactions are fraudulent may be more sensitive to false negatives than to false positives. To promote customer satisfaction, the credit card company may prefer to risk verifying legitimate transactions than risk ignoring a fraudulent transaction. Because most transactions are legitimate, accuracy is not an appropriate metric for this problem. A classifier that always predicts that transactions are legitimate could have a high accuracy score, but would not be useful. For these reasons, classifiers are

often evaluated using two additional measures called precision and recall.

Precision and recall

Recall from [Chapter 1](#), *The Fundamentals of Machine Learning*, that precision is the fraction of positive predictions that are correct. For instance, in our SMS spam classifier, precision is the fraction of messages classified as spam that are actually spam. Precision is given by the following ratio:

$$P = \frac{TP}{TP + FP}$$

Sometimes called sensitivity in medical domains, recall is the fraction of the truly positive instances that the classifier recognizes. A recall score of one indicates that the classifier did not make any false negative predictions. For our SMS spam classifier, recall is the fraction of spam messages that were truly classified as spam. Recall is calculated with the following ratio:

$$R = \frac{TP}{TP + FN}$$

Individually, precision and recall are seldom informative; they are both incomplete views of a classifier's performance. Both precision and recall can fail to distinguish classifiers that perform well from certain types of classifiers that perform poorly. A trivial classifier could easily achieve a perfect recall score by predicting positive for every instance. For example, assume that a test set contains ten positive examples and ten negative examples. A classifier that predicts positive for every example will achieve a recall of one, as follows:

$$R = \frac{10}{10+0} = 1$$

A classifier that predicts negative for every example, or that makes only false positive and true negative predictions, will achieve a recall score of zero. Similarly, a classifier that predicts that only a single instance is positive and happens to be correct will achieve perfect precision.

scikit-learn provides a function to calculate the precision and recall for a classifier from a set of predictions and the corresponding set of trusted labels. Let's calculate our SMS classifier's precision and recall:

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import
TfidfVectorizer
>>> from sklearn.linear_model.logistic import
LogisticRegression
>>> from sklearn.cross_validation import
train_test_split, cross_val_score
>>> df = pd.read_csv('data/sms.csv')
>>> X_train_raw, X_test_raw, y_train, y_test =
train_test_split(df['message'], df['label'])
>>> vectorizer = TfidfVectorizer()
>>> X_train =
vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> precisions = cross_val_score(classifier,
X_train, y_train, cv=5, scoring='precision')
>>> print 'Precision', np.mean(precisions),
precisions
>>> recalls = cross_val_score(classifier,
X_train, y_train, cv=5, scoring='recall')
>>> print 'Recalls', np.mean(recalls), recalls

Precision 0.992137651822 [ 0.98717949
0.98666667 1. 0.98684211 1. ]
Recall 0.677114261885 [ 0.7 0.67272727
0.6 0.68807339 0.72477064]
```

Our classifier's precision is 0.992; almost all of the messages that it predicted as spam were actually spam. Its recall is lower, indicating that it incorrectly classified

approximately 22 percent of the spam messages as ham. Your precision and recall may vary since the training and test data are randomly partitioned.

Calculating the F1 measure

The F1 measure is the harmonic mean, or weighted average, of the precision and recall scores. Also called the f-measure or the f-score, the F1 score is calculated using the following formula:

$$F1 = 2 \frac{PR}{P+R}$$

The F1 measure penalizes classifiers with imbalanced precision and recall scores, like the trivial classifier that always predicts the positive class. A model with perfect precision and recall scores will achieve an F1 score of one. A model with a perfect precision score and a recall score of zero will achieve an F1 score of zero. As for precision and recall, scikit-learn provides a function to calculate the F1 score for a set of predictions. Let's compute our classifier's F1 score. The following snippet continues the previous code sample:

```
>>> f1s = cross_val_score(classifier, x_train,  
y_train, cv=5, scoring='f1')
```

```
>>> print 'F1', np.mean(f1s), f1s  
  
F1 0.80261302628 [ 0.82539683  0.8  
0.77348066  0.83157895  0.7826087 ]
```

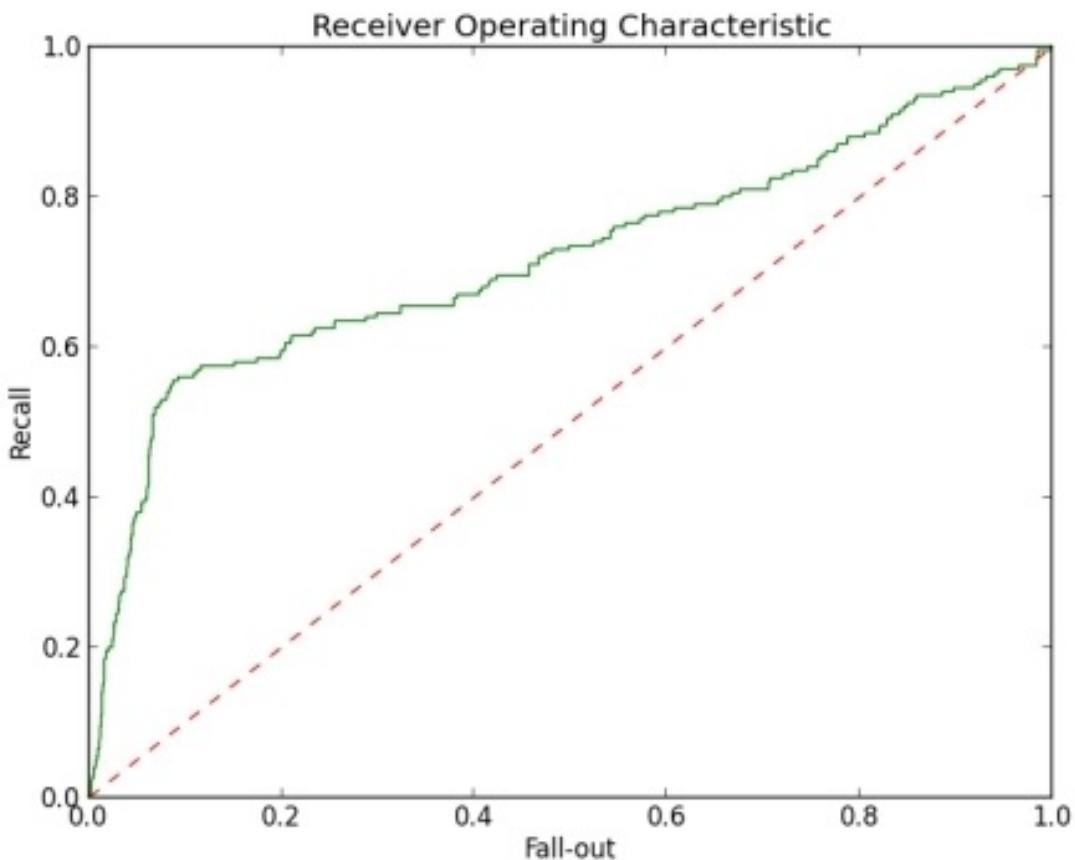
The arithmetic mean of our classifier's precision and recall scores is 0.803. As the difference between the classifier's precision and recall is small, the F1 measure's penalty is small. Models are sometimes evaluated using the F0.5 and F2 scores, which favor precision over recall and recall over precision, respectively.

ROC AUC

A **Receiver Operating Characteristic**, or **ROC curve**, visualizes a classifier's performance. Unlike accuracy, the ROC curve is insensitive to data sets with unbalanced class proportions; unlike precision and recall, the ROC curve illustrates the classifier's performance for all values of the discrimination threshold. ROC curves plot the classifier's recall against its **fall-out**. Fall-out, or the false positive rate, is the number of false positives divided by the total number of negatives. It is calculated using the following formula:

$$F = \frac{FP}{TN + FP}$$

AUC is the area under the ROC curve; it reduces the ROC curve to a single value, which represents the expected performance of the classifier. The dashed line in the following figure is for a classifier that predicts classes randomly; it has an AUC of 0.5. The solid curve is for a classifier that outperforms random guessing:



Let's plot the ROC curve for our SMS spam classifier:

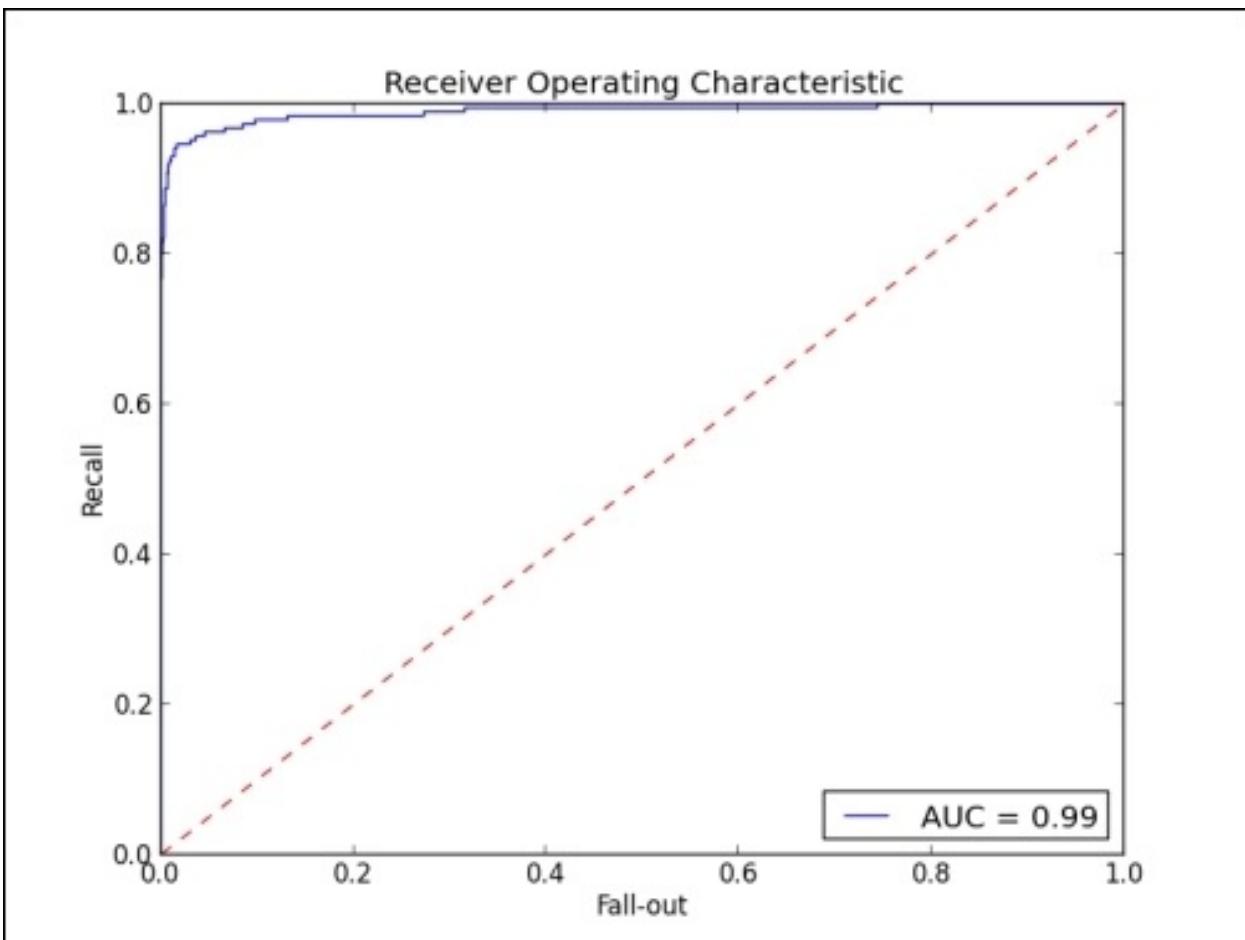
```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> from sklearn.feature_extraction.text import
TfidfVectorizer
>>> from sklearn.linear_model.logistic import
LogisticRegression
>>> from sklearn.cross_validation import
train_test_split, cross_val_score
```

```

>>> from sklearn.metrics import roc_curve, auc
>>> df = pd.read_csv('data/sms.csv')
>>> X_train_raw, X_test_raw, y_train, y_test =
train_test_split(df['message'], df['label'])
>>> vectorizer = TfidfVectorizer()
>>> X_train =
vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> predictions =
classifier.predict_proba(X_test)
>>> false_positive_rate, recall, thresholds =
roc_curve(y_test, predictions[:, 1])
>>> roc_auc = auc(false_positive_rate, recall)
>>> plt.title('Receiver Operating
Characteristic')
>>> plt.plot(false_positive_rate, recall, 'b',
label='AUC = %.2f' % roc_auc)
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1], 'r--')
>>> plt.xlim([0.0, 1.0])
>>> plt.ylim([0.0, 1.0])
>>> plt.ylabel('Recall')
>>> plt.xlabel('Fall-out')
>>> plt.show()

```

From the ROC AUC plot, it is apparent that our classifier outperforms random guessing; most of the plot area lies under its curve:



Tuning models with grid search

Hyperparameters are parameters of the model that are not learned. For example, hyperparameters of our logistic regression SMS classifier include the value of the regularization term and thresholds used to remove words that appear too frequently or infrequently. In scikit-learn, hyperparameters are set through the model's constructor. In the previous examples, we did not set any arguments for `LogisticRegression()`; we used the default values for all of the hyperparameters. These default values are often a good start, but they may not produce the optimal model. **Grid search** is a common method to select the hyperparameter values that produce the best model. Grid search takes a set of possible values for each hyperparameter that should be tuned, and evaluates a model trained on each element of the Cartesian product of the sets. That is, grid search is an exhaustive search that trains and evaluates a model for each possible combination of the hyperparameter values supplied by the developer. A disadvantage of grid search is that it is computationally costly for even small sets of hyperparameter values. Fortunately, it is an **embarrassingly parallel** problem; many models can

easily be trained and evaluated concurrently since no synchronization is required between the processes. Let's use scikit-learn's `GridSearchCV()` function to find better hyperparameter values:

```
import pandas as pd
from sklearn.feature_extraction.text import
TfidfVectorizer
from sklearn.linear_model.logistic import
LogisticRegression
from sklearn.grid_search import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.cross_validation import
train_test_split
from sklearn.metrics import precision_score,
recall_score, accuracy_score

pipeline = Pipeline([
    ('vect',
    TfidfVectorizer(stop_words='english')),
    ('clf', LogisticRegression())
])
parameters = {
    'vect__max_df': (0.25, 0.5, 0.75),
    'vect__stop_words': ('english', None),
    'vect__max_features': (2500, 5000, 10000,
None),
    'vect__ngram_range': ((1, 1), (1, 2)),
    'vect__use_idf': (True, False),
    'vect__norm': ('l1', 'l2'),
    'clf__penalty': ('l1', 'l2'),
    'clf__C': (0.01, 0.1, 1, 10),
}
```

`GridSearchCV()` takes an estimator, a parameter space, and performance measure. The argument `n_jobs` specifies the maximum number of concurrent jobs; set `n_jobs` to `-1` to use all CPU cores. Note that `fit()` must be called in a Python `main` block in order to fork additional processes; this example must be executed as a script, and not in an interactive interpreter:

```
if __name__ == "__main__":
    grid_search = GridSearchCV(pipeline,
parameters, n_jobs=-1, verbose=1,
scoring='accuracy', cv=3)
    df = pd.read_csv('data/sms.csv')
    X, y, = df['message'], df['label']
    X_train, X_test, y_train, y_test =
train_test_split(X, y)
    grid_search.fit(X_train, y_train)
    print 'Best score: %0.3f' %
grid_search.best_score_
    print 'Best parameters set:'
    best_parameters =
grid_search.best_estimator_.get_params()
    for param_name in sorted(parameters.keys()):
        print '\t%s: %r' % (param_name,
best_parameters[param_name])
    predictions = grid_search.predict(X_test)
    print 'Accuracy:', accuracy_score(y_test,
predictions)
    print 'Precision:', precision_score(y_test,
predictions)
    print 'Recall:', recall_score(y_test,
predictions)
```

The following is the output of the script:

```
Fitting 3 folds for each of 1536 candidates,
totalling 4608 fits
[Parallel(n_jobs=-1)]: Done    1 jobs          |
elapsed: 0.2s
[Parallel(n_jobs=-1)]: Done   50 jobs          |
elapsed: 4.0s
[Parallel(n_jobs=-1)]: Done  200 jobs          |
elapsed: 16.9s
[Parallel(n_jobs=-1)]: Done  450 jobs          |
elapsed: 36.7s
[Parallel(n_jobs=-1)]: Done  800 jobs          |
elapsed: 1.1min
[Parallel(n_jobs=-1)]: Done 1250 jobs          |
elapsed: 1.7min
[Parallel(n_jobs=-1)]: Done 1800 jobs          |
elapsed: 2.5min
[Parallel(n_jobs=-1)]: Done 2450 jobs          |
elapsed: 3.4min
[Parallel(n_jobs=-1)]: Done 3200 jobs          |
elapsed: 4.4min
[Parallel(n_jobs=-1)]: Done 4050 jobs          |
elapsed: 7.7min
[Parallel(n_jobs=-1)]: Done 4608 out of 4608  |
elapsed: 8.5min finished
Best score: 0.983
Best parameters set:
  clf__C: 10
  clf__penalty: 'l2'
  vect__max_df: 0.5
  vect__max_features: None
  vect__ngram_range: (1, 2)
  vect__norm: 'l2'
```

```
vect_stop_words: None  
vect_use_idf: True  
Accuracy: 0.989956958393  
Precision: 0.988095238095  
Recall: 0.932584269663
```

Optimizing the values of the hyperparameters has improved our model's recall score on the test set.

Multi-class classification

In the previous sections you learned to use logistic regression for binary classification. In many classification problems, however, there are more than two classes that are of interest. We might wish to predict the genres of songs from samples of audio, or classify images of galaxies by their types. The goal of **multi-class classification** is to assign an instance to one of the set of classes. scikit-learn uses a strategy called **one-vs.-all**, or **one-vs.-the-rest**, to support multi-class classification. One-vs.-all classification uses one binary classifier for each of the possible classes. The class that is predicted with the greatest confidence is assigned to the instance. LogisticRegression supports multi-class classification using the one-versus-all strategy out of the box. Let's use LogisticRegression for a multi-class classification problem.

Assume that you would like to watch a movie, but you have a strong aversion to watching bad movies. To inform your decision, you could read reviews of the movies you are considering, but unfortunately you also have a strong aversion to reading movie reviews. Let's use scikit-learn to find the movies with good reviews.

In this example, we will classify the sentiments of phrases taken from movie reviews in the Rotten Tomatoes data set. Each phrase can be classified as one of the following sentiments: negative, somewhat negative, neutral, somewhat positive, or positive. While the classes appear to be ordered, the explanatory variables that we will use do not always corroborate this order due to sarcasm, negation, and other linguistic phenomena. Instead, we will approach this problem as a multi-class classification task.

The data can be downloaded from <http://www.kaggle.com/c/sentiment-analysis-on-movie-reviews/data>. First, let's explore the data set using pandas. Note that the import and data-loading statements in the following snippet are required for the subsequent snippets:

```
>>> import pandas as pd  
>>> df = pd.read_csv('movie-reviews/train.tsv',  
header=0, delimiter='\t')  
>>> print df.count()  
  
PhraseId      156060  
SentenceId    156060  
Phrase         156060  
Sentiment      156060  
dtype: int64
```

The columns of the data set are tab delimited. The data set

contains 1,56,060 instances.

```
>>> print df.head()
```

```
PhraseId SentenceId
Phrase \
0           1           1   A series of escapades
demonstrating the adage ...
1           2           1   A series of escapades
demonstrating the adage ...
2           3           1
A series
3           4           1
A
4           5           1
series

Sentiment
0           1
1           2
2           2
3           2
4           2

[5 rows x 4 columns]
```

The `Sentiment` column contains the response variables. The 0 label corresponds to the sentiment negative, 1 corresponds to somewhat negative, and so on. The `Phrase` column contains the raw text. Each sentence from the movie reviews has been parsed into smaller phrases. We will not require the `PhraseId` and `SentenceId`

columns in this example. Let's print some of the phrases and examine them:

```
>>> print df['Phrase'].head(10)

0    A series of escapades demonstrating the
adage ...
1    A series of escapades demonstrating the
adage ...
2                                A
series
3
A
4
series
5    of escapades demonstrating the adage that
what...
6
of
7    escapades demonstrating the adage that what
is...
8
escapades
9    demonstrating the adage that what is good
for ...
Name: Phrase, dtype: object
```

Now let's examine the target classes:

```
>>> print df['Sentiment'].describe()
```

count	156060.000000
mean	2.063578

```
std          0.893832
min          0.000000
25%         2.000000
50%         2.000000
75%         3.000000
max          4.000000
Name: Sentiment, dtype: float64
```

```
>>> print df['Sentiment'].value_counts()
```

```
2    79582
3    32927
1    27273
4    9206
0    7072
dtype: int64
```

```
>>> print
df['Sentiment'].value_counts()/df['Sentiment'].c
ount()
```

```
2    0.509945
3    0.210989
1    0.174760
4    0.058990
0    0.045316
dtype: float64
```

The most common class, `Neutral`, includes more than 50 percent of the instances. Accuracy will not be an informative performance measure for this problem, as a degenerate classifier that predicts only `Neutral` can obtain

an accuracy near 0.5. Approximately one quarter of the reviews are positive or somewhat positive, and approximately one fifth of the reviews are negative or somewhat negative. Let's train a classifier with scikit-learn:

```
import pandas as pd
from sklearn.feature_extraction.text import
TfidfVectorizer
from sklearn.linear_model.logistic import
LogisticRegression
from sklearn.cross_validation import
train_test_split
from sklearn.metrics import
classification_report, accuracy_score,
confusion_matrix
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV

def main():
    pipeline = Pipeline([
        ('vect',
         TfidfVectorizer(stop_words='english')),
        ('clf', LogisticRegression())
    ])
    parameters = {
        'vect_max_df': (0.25, 0.5),
        'vect_ngram_range': ((1, 1), (1, 2)),
        'vect_use_idf': (True, False),
        'clf_C': (0.1, 1, 10),
    }
    df = pd.read_csv('data/train.tsv', header=0,
```

```

delimiter='\t')
    X, y = df['Phrase'],
df['Sentiment'].as_matrix()
    X_train, X_test, y_train, y_test =
train_test_split(X, y, train_size=0.5)
    grid_search = GridSearchCV(pipeline,
parameters, n_jobs=3, verbose=1,
scoring='accuracy')
    grid_search.fit(X_train, y_train)
    print 'Best score: %0.3f' %
grid_search.best_score_
    print 'Best parameters set:'
    best_parameters =
grid_search.best_estimator_.get_params()
    for param_name in sorted(parameters.keys()):
        print '\t%s: %r' % (param_name,
best_parameters[param_name])

if __name__ == '__main__':
    main()

```

The following is the output of the script:

```

Fitting 3 folds for each of 24 candidates,
totalling 72 fits
[Parallel(n_jobs=3)]: Done   1 jobs          |
elapsed:  3.3s
[Parallel(n_jobs=3)]: Done   50 jobs         |
elapsed: 1.1min
[Parallel(n_jobs=3)]: Done   68 out of  72 |
elapsed: 1.9min remaining: 6.8s
[Parallel(n_jobs=3)]: Done   72 out of  72 |
elapsed: 2.1min finished

```

Best score: 0.620

Best parameters set:

clf_C: 10

vect_max_df: 0.25

vect_ngram_range: (1, 2)

vect_use_idf: False

Multi-class classification performance metrics

As with binary classification, confusion matrices are useful for visualizing the types of errors made by the classifier. Precision, recall, and F1 score can be computed for each of the classes, and accuracy for all of the predictions can also be calculated. Let's evaluate our classifier's predictions. The following snippet continues the previous example:

```
predictions = grid_search.predict(X_test)
print 'Accuracy:', accuracy_score(y_test,
predictions)
print 'Confusion Matrix:',
confusion_matrix(y_test, predictions)
print 'Classification Report:',
classification_report(y_test, predictions)
```

The following will be appended to the output:

```
Accuracy: 0.636370626682
Confusion Matrix: [[ 1129   1679    634     64
9]
 [ 917   6121   6084    505     35]
 [ 229   3091   32688   3614    166]
 [  34    408   6734   8068   1299]
 [   5     35    494   2338   1650]]
Classification Report:               precision
recall  f1-score   support
```

	0	0.49	0.32	0.39
3515	1	0.54	0.45	0.49
13662	2	0.70	0.82	0.76
39788	3	0.55	0.49	0.52
16543	4	0.52	0.36	0.43
4522				
avg / total		0.62	0.64	0.62
78030				

First, we make predictions using the best parameter set found by using grid searching. While our classifier is an improvement over the baseline classifier, it frequently mistakes Somewhat Positive and Somewhat Negative for Neutral.

Multi-label classification and problem transformation

In the previous sections, we discussed binary classification, in which each instance must be assigned to one of the two classes, and multi-class classification, in which each instance must be assigned to one of the set of classes. The final type of classification problem that we will discuss is multi-label classification, in which each instance can be assigned a subset of the set of classes. Examples of multi-label classification include assigning tags to messages posted on a forum, and classifying the objects present in an image. There are two groups of approaches for multi-label classification.

Problem transformation methods are techniques that cast the original multi-label problem as a set of single-label classification problems. The first problem transformation method that we will review converts each set of labels encountered in the training data to a single label. For example, consider a multi-label classification problem in which news articles must be assigned to one or more categories from a set. The following training data

contains seven articles that can pertain to one or more of the five categories.

Instance	Categories				
	Local	US	Business	Science and Technology	Sports
1					
2					
3					
4					
5					
6					
7					

Transforming the problem into a single-label classification task using the power set of labels seen in the training data results in the following training data. Previously, the first instance was classified as Local and US. Now it has a single label, Local \wedge US.

	Category						
Instance	Local	Local ^ US	Business	Local ^ Business	US ^ Science and Technology	Business ^ Science and Technology	Sport
1							
2							
3							
4							
5							
6							
7							

The multi-label classification problem that had five classes is now a multi-class classification problem with seven classes. While the power set problem transformation is intuitive, increasing the number of classes is frequently impractical; this transformation can

produce many new labels that correspond to only a few training instances. Furthermore, the classifier can only predict combinations of labels that were seen in the training data.

	Category			Category	
Instance	Local	\neg Local	Instance	Business	\neg Business
1			1		
2			2		
3			3		
4			4		
5			5		
6			6		
7			7		
	Category			Category	

Instance	US	\neg US	Instance	US	\neg US
1			1		
2			2		
3			3		
4			4		
5			5		
6			6		
7			7		
	Category			Category	
Instance	Sci. and Tech.	\neg Sci. and Tech.	Instance	Sports	\neg Sports
1			1		
2			2		
3			3		

	4			4		
5				5		
6				6		

A second problem transformation is to train one binary classifier for each of the labels in the training set. Each classifier predicts whether or not the instance belongs to one label. Our example would require five binary classifiers; the first classifier would predict whether or not an instance should be classified as `Local`, the second classifier would predict whether or not an instance should be classified as `US`, and so on. The final prediction is the union of the predictions from all of the binary classifiers. The transformed training data is shown in the previous figure. This problem transformation ensures that the single-label problems will have the same number of training examples as the multilabel problem, but ignores relationships between the labels.

Multi-label classification performance metrics

Multi-label classification problems must be assessed using different performance measures than single-label classification problems. Two of the most common performance metrics are **Hamming loss** and **Jaccard similarity**. Hamming loss is the average fraction of incorrect labels. Note that Hamming loss is a loss function, and that the perfect score is zero. Jaccard similarity, or the Jaccard index, is the size of the intersection of the predicted labels and the true labels divided by the size of the union of the predicted and true labels. It ranges from zero to one, and one is the perfect score. Jaccard similarity is calculated by the following equation:

$$J(\text{Predicted}, \text{True}) = \frac{|\text{Predicted} \cap \text{True}|}{|\text{Predicted} \cup \text{True}|}$$

```
>>> import numpy as np
>>> from sklearn.metrics import hamming_loss
>>> print hamming_loss(np.array([[0.0, 1.0],
[1.0, 1.0]]), np.array([[0.0, 1.0], [1.0,
1.0]]))
```

```
0.0
>>> print hamming_loss(np.array([[0.0, 1.0],
[1.0, 1.0]]), np.array([[1.0, 1.0], [1.0,
1.0]]))
0.25
>>> print hamming_loss(np.array([[0.0, 1.0],
[1.0, 1.0]]), np.array([[1.0, 1.0], [0.0,
1.0]]))
0.5
>>> print
jaccard_similarity_score(np.array([[0.0, 1.0],
[1.0, 1.0]]), np.array([[0.0, 1.0], [1.0,
1.0]]))
1.0
>>> print
jaccard_similarity_score(np.array([[0.0, 1.0],
[1.0, 1.0]]), np.array([[1.0, 1.0], [1.0,
1.0]]))
0.75
>>> print
jaccard_similarity_score(np.array([[0.0, 1.0],
[1.0, 1.0]]), np.array([[1.0, 1.0], [0.0,
1.0]]))
0.5
```

Summary

In this chapter we discussed generalized linear models, which extend ordinary linear regression to support response variables with non-normal distributions.

Generalized linear models use a link function to relate a linear combination of the explanatory variables to the response variable; unlike ordinary linear regression, the relationship does not need to be linear. In particular, we examined the logistic link function, a sigmoid function that returns a value between zero and one for any real number.

We discussed logistic regression, a generalized linear model that uses the logistic link function to relate explanatory variables to a Bernoulli-distributed response variable. Logistic regression can be used for binary classification, a task in which an instance must be assigned to one of the two classes; we used logistic regression to classify spam and ham SMS messages. We then discussed multi-class classification, a task in which each instance must be assigned one label from a set of labels. We used the one-vs.-all strategy to classify the sentiments of movie reviews. Finally, we discussed multi-label classification, in which instances must be assigned a subset of a set of labels. Having completed our discussion

of regression and classification with generalized linear models, we will introduce a non-linear model for regression and classification called the decision tree in the next chapter.

Chapter 5. Nonlinear Classification and Regression with Decision Trees

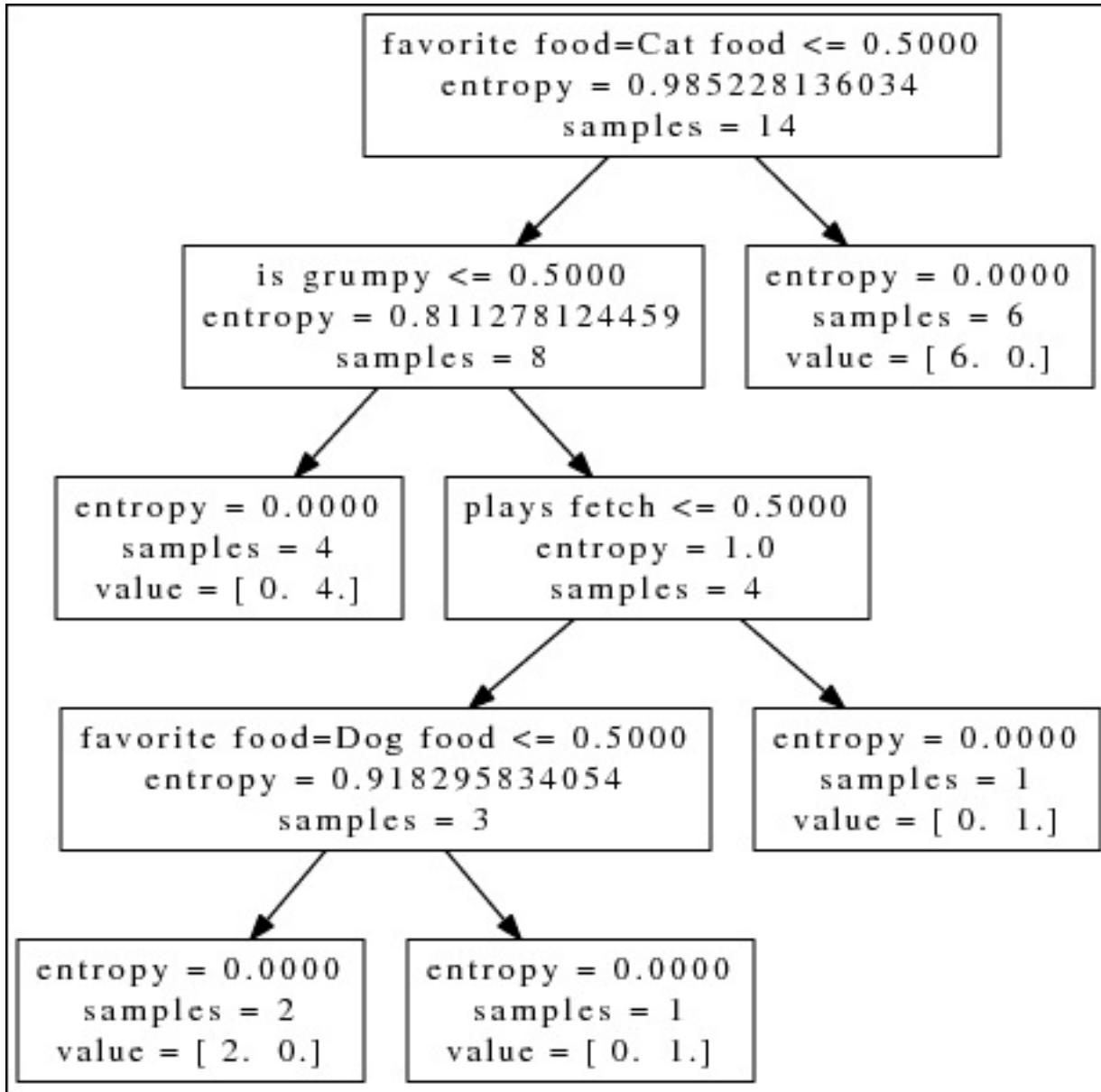
In the previous chapters we discussed generalized linear models, which relate a linear combination of explanatory variables to one or more response variables using a link function. You learned to use multiple linear regression to solve regression problems, and we used logistic regression for classification tasks. In this chapter we will discuss a simple, nonlinear model for classification and regression tasks: the decision tree. We'll use decision trees to build an ad blocker that can learn to classify images on a web page as banner advertisements or page content. Finally, we will introduce ensemble learning methods, which combine a set of models to produce an estimator with better predictive performance than any of its component estimators.

Decision trees

Decision trees are tree-like graphs that model a decision. They are analogous to the parlor game Twenty Questions. In Twenty Questions, one player, called the answerer, chooses an object but does not reveal the object to the other players, who are called questioners. The object should be a common noun, such as "guitar" or "sandwich", but not "1969 Gibson Les Paul Custom" or "North Carolina". The questioners must guess the object by asking as many as twenty questions that can be answered with yes, no, or maybe. An intuitive strategy for questioners is to ask questions of increasing specificity; asking "*is it a musical instrument?*" as the first question will not efficiently reduce the number of possibilities. The branches of a decision tree specify the shortest sequences of explanatory variables that can be examined in order to estimate the value of a response variable. To continue the analogy, in Twenty Questions the questioner and the answerers all have knowledge of the training data, but only the answerer knows the values of the features for the test instance.

Decision trees are commonly learned by recursively splitting the set of training instances into subsets based on the instances' values for the explanatory variables. The

following diagram depicts a decision tree that we will look at in more detail later in the chapter.



Represented by boxes, the interior nodes of the decision tree test explanatory variables. These nodes are connected

by edges that specify the possible outcomes of the tests. The training instances are divided into subsets based on the outcomes of the tests. For example, a node might test whether or not the value of an explanatory variable exceeds a threshold. The instances that pass the test will follow an edge to the node's right child, and the instances that fail the test will follow an edge to the node's left child. The children nodes similarly test their subsets of the training instances until a stopping criterion is satisfied. In classification tasks, the leaf nodes of the decision tree represent classes. In regression tasks, the values of the response variable for the instances contained in a leaf node may be averaged to produce the estimate for the response variable. After the decision tree has been constructed, making a prediction for a test instance requires only following the edges until a leaf node is reached.

Training decision trees

Let's create a decision tree using an algorithm called **Iterative Dichotomiser 3 (ID3)**. Invented by Ross Quinlan, ID3 was one of the first algorithms used to train decision trees. Assume that you have to classify animals as cats or dogs. Unfortunately, you cannot observe the animals directly and must use only a few attributes of the animals to make your decision. For each animal, you are told whether or not it likes to play fetch, whether or not it is frequently grumpy, and its favorite of three types of food.

To classify new animals, the decision tree will examine an explanatory variable at each node. The edge it follows to the next node will depend on the outcome of the test. For example, the first node might ask whether or not the animal likes to play fetch. If the animal does, we will follow the edge to the left child node; if not, we will follow the edge to the right child node. Eventually an edge will connect to a leaf node that indicates whether the animal is a cat or a dog.

The following fourteen instances comprise our training data:



Training instance	Plays fetch	Is grumpy	Favorite food	Species
1	Yes	No	Bacon	Dog
2	No	Yes	Dog Food	Dog
3	No	Yes	Cat food	Cat
4	No	Yes	Bacon	Cat
5	No	No	Cat food	Cat
6	No	Yes	Bacon	Cat
7	No	Yes	Cat Food	Cat
8	No	No	Dog Food	Dog
9	No	Yes	Cat food	Cat
10	Yes	No	Dog Food	Dog
11	Yes	No	Bacon	Dog
12	No	No	Cat food	Cat

13	Yes	Yes	Cat food	Cat
14	Yes	Yes	Bacon	Dog

From this data we can see that cats are generally grumpier than the dogs. Most dogs play fetch and most cats refuse. Dogs prefer dog food and bacon, whereas cats only like cat food and bacon. The `is_grumpy` and `plays_fetch` explanatory variables can be easily converted to binary-valued features. The `favorite_food` explanatory variable is a categorical variable that has three possible values; we will one-hot encode it. Recall from [Chapter 3, Feature Extraction and Preprocessing](#), that one-hot encoding represents a categorical variable with as many binary-valued features as there are values for variable.

Representing the categorical variable with a single integer-valued feature will encode an artificial order to its values. Since `favorite_food` has three possible states, we will represent it with three binary-valued features. From this table, we can manually construct classification rules. For example, an animal that is grumpy and likes cat food must be a cat, while an animal that plays fetch and likes bacon must be a dog. Constructing these classification rules by hand for even a small data set is cumbersome. Instead, we will learn these rules by creating a decision tree.

Selecting the questions

Like Twenty Questions, the decision tree will estimate the value of the response variable by testing the values of a sequence of explanatory variables. Which explanatory variable should be tested first? Intuitively, a test that produces subsets that contain all cats or all dogs is better than a test that produces subsets that still contain both cats and dogs. If the members of a subset are of different classes, we are still uncertain about how to classify the instance. We should also avoid creating tests that separate only a single cat or dog from the others; such tests are analogous to asking specific questions in the first few rounds of Twenty Questions. More formally, these tests can infrequently classify an instance and might not reduce our uncertainty. The tests that reduce our uncertainty about the classification the most are the best. We can quantify the amount of uncertainty using a measure called **entropy**.

Measured in bits, entropy quantifies the amount of uncertainty in a variable. Entropy is given by the following equation, where n is the number of outcomes and $P(x_i)$ is the probability of the outcome i . Common values for b are 2 , e , and 10 . Because the log of a number less than one will be negative, the entire sum is negated to

return a positive value.

$$H(X) = -\sum_{i=1}^n P(x_i) \log_b P(x_i)$$

For example, a single toss of a fair coin has only two outcomes: heads and tails. The probability that the coin will land on heads is 0.5, and the probability that it will land on tails is 0.5. The entropy of the coin toss is equal to the following:

$$H(X) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1.0$$

That is, only one bit is required to represent the two equally probable outcomes, heads and tails. Two tosses of a fair coin can result in four possible outcomes: heads and heads, heads and tails, tails and heads, and tails and tails. The probability of each outcome is $0.5 \times 0.5 = 0.25$. The entropy of two tosses is equal to the following:

$$H(X) = -(0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25) = 2.0$$

If the coin has the same face on both sides, the variable representing its outcome has 0 bits of entropy; that is, we are always certain of the outcome and the variable will never represent new information. Entropy can also be represented as a fraction of a bit. For example, an unfair coin has two different faces, but is weighted such that the faces are not equally likely to land in a toss. Assume that the probability that an unfair coin will land on heads is 0.8, and the probability that it will land on tails is 0.2. The entropy of a single toss of this coin is equal to the following:

$$H(X) = -(0.9 \log_2 0.9 + 0.2 \log_2 0.2) = 0.7219280948873623$$

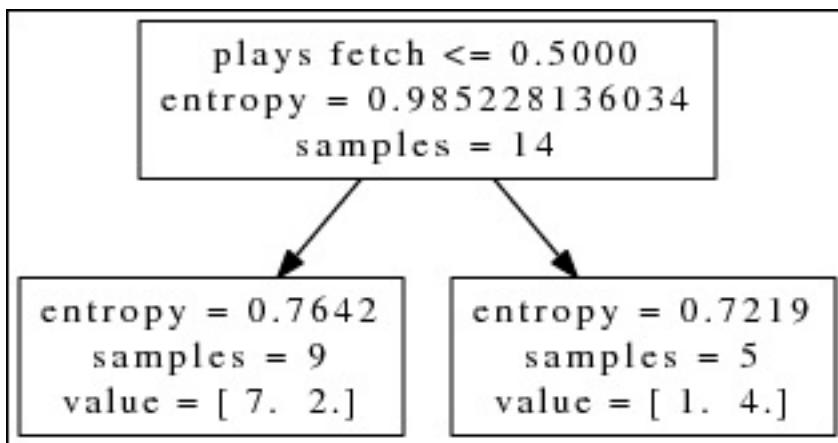
The outcome of a single toss of an unfair coin can have a fraction of one bit of entropy. There are two possible outcomes of the toss, but we are not totally uncertain since one outcome is more frequent.

Let's calculate the entropy of classifying an unknown animal. If an equal number of dogs and cats comprise our animal classification training data and we do not know anything else about the animal, the entropy of the decision is equal to one. All we know is that the animal could be either a cat or a dog; like the fair coin toss, both outcomes

are equally likely. Our training data, however, contains six dogs and eight cats. If we do not know anything else about the unknown animal, the entropy of the decision is given by the following:

$$H(X) = -\left(\frac{6}{14} \log_2 \frac{6}{14} + \frac{8}{14} \log_2 \frac{8}{14} \right) = 0.985228136.342516$$

Since cats are more common, we are less uncertain about the outcome. Now let's find the explanatory variable that will be most helpful in classifying the animal; that is, let's find the explanatory variable that reduces the entropy the most. We can test the `plays fetch` explanatory variable and divide the training instances into animals that play fetch and animals that don't. This produces the two following subsets:



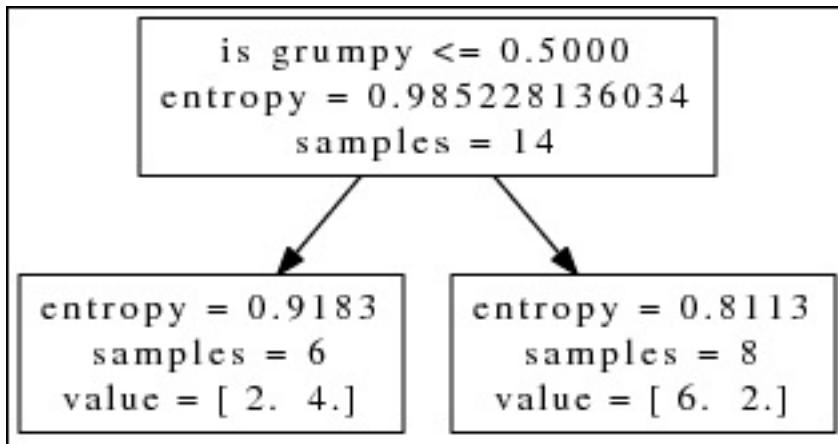
Decision trees are often visualized as diagrams that are similar to flowcharts. The top box of the previous diagram is the root node; it contains all of our training instances and specifies the explanatory variable that will be tested. At the root node we have not eliminated any instances from the training set and the entropy is equal to approximately 0.985. The root node tests the `plays fetch` explanatory variable. Recall that we converted this Boolean explanatory variable to a binary-valued feature. Training instances for which `plays fetch` is equal to zero follow the edge to the root's left child, and training instances for animals that do play fetch follow the edge to the root's right child node. The left child node contains a subset of the training data with seven cats and two dogs that do not like to play fetch. The entropy at this node is given by the following:

$$H(X) = -\left(\frac{2}{9} \log_2 \frac{2}{9} + \frac{7}{9} \log_2 \frac{7}{9}\right) = 0.7642045065086203$$

The right child contains a subset with one cat and four dogs that do like to play fetch. The entropy at this node is given by the following:

$$H(X) = -\left(\frac{1}{5} \log_2 \frac{1}{5} + \frac{4}{5} \log_2 \frac{4}{5}\right) = 0.7219280948873623$$

Instead of testing the `plays fetch` explanatory variable, we could test the `is grumpy` explanatory variable. This test produces the following tree. As with the previous tree, instances that fail the test follow the left edge, and instances that pass the test follow the right edge.



We could also divide the instances into animals that prefer cat food and animals that don't to produce the following tree:

```
favorite food=cat food <= 0.5000
entropy = 0.985228136034
samples = 14
```

```
entropy = 0.8113
samples = 8
value = [ 2.  6.]
```

```
entropy = 0.0000
samples = 6
value = [ 6.  0.]
```

Information gain

Testing for the animals that prefer cat food resulted in one subset with six cats, zero dogs, and 0 bits of entropy and another subset with two cats, six dogs, and 0.811 bits of entropy. How can we measure which of these tests reduced our uncertainty about the classification the most? Averaging the entropies of the subsets may seem to be an appropriate measure of the reduction in entropy. In this example, the subsets produced by the cat food test have the lowest average entropy. Intuitively, this test seems to be effective, as we can use it to classify almost half of the training instances. However, selecting the test that produces the subsets with the lowest average entropy can produce a suboptimal tree. For example, imagine a test that produced one subset with two dogs and no cats and another subset with four dogs and eight cats. The entropy of the first subset is equal to the following (note that the second term is omitted because $\log_2 0$ is undefined):

$$H(X) = -\left(\frac{2}{2} \log_2 \frac{2}{2}\right) = 0.0$$

The entropy of the second subset is equal to the

following:

$$H(X) = -\left(\frac{4}{12} \log_2 \frac{4}{12} + \frac{8}{12} \log_2 \frac{8}{12} \right) = 0.9182958340544896$$

The average of these subsets' entropies is only 0.459, but the subset containing most of the instances has almost one bit of entropy. This is analogous to asking specific questions early in Twenty Questions; we could get lucky and win within the first few attempts, but it is more likely that we will squander our questions without eliminating many possibilities. Instead, we will measure the reduction in entropy using a metric called **information gain**. Calculated with the following equation, information gain is the difference between the entropy of the parent node, $H(T)$, and the weighted average of the children nodes' entropies. T is the set of instances, and a is the explanatory variable under test. $x_a \in vals(a)$ is the value of attribute a for instance x . $\{x \in T \mid x_a = v\}$ is the number of instances for which attribute a is equal to the value v . $H(\{x \in T \mid x_a = v\})$ is the entropy of the subset of instances for which the value of the explanatory variable a is v .

$$IG(T, a) = H(T) - \sum_{v \in vals(a)} \frac{|\{x \in T \mid x_a = v\}|}{|T|} H(\{x \in T \mid x_a = v\})$$

The following table contains the information gains for all of the tests. In this case, the cat food test is still the best, as it increases the information gain the most.

Test	Parent's entropy	Child's entropy	Child's entropy	Weighted average	IG
plays fetch?	0.9852	0.7642	0.7219	$0.7490 * 9/14 + 0.7219 * 5/14 = 0.7491$	0.2361
is grumpy?	0.9852	0.9183	0.8113	$0.9183 * 6/14 + 0.8113 * 8/14 = 0.8571$ 0.8572	0.1280
favorite food = cat food	0.9852	0.8113	0	$0.8113 * 8/14 + 0.0 * 6/14 = 0.4636$	0.5216
favorite food = dog food	0.9852	0.8454	0	$0.8454 * 11/14 + 0.0 * 3/14 = 0.6642$	0.3210
favorite food =	0.9852	0.9183	0.971	$0.9183 * 9/14 + 0.9710 * 5/14 = 0.9710$	0.0481

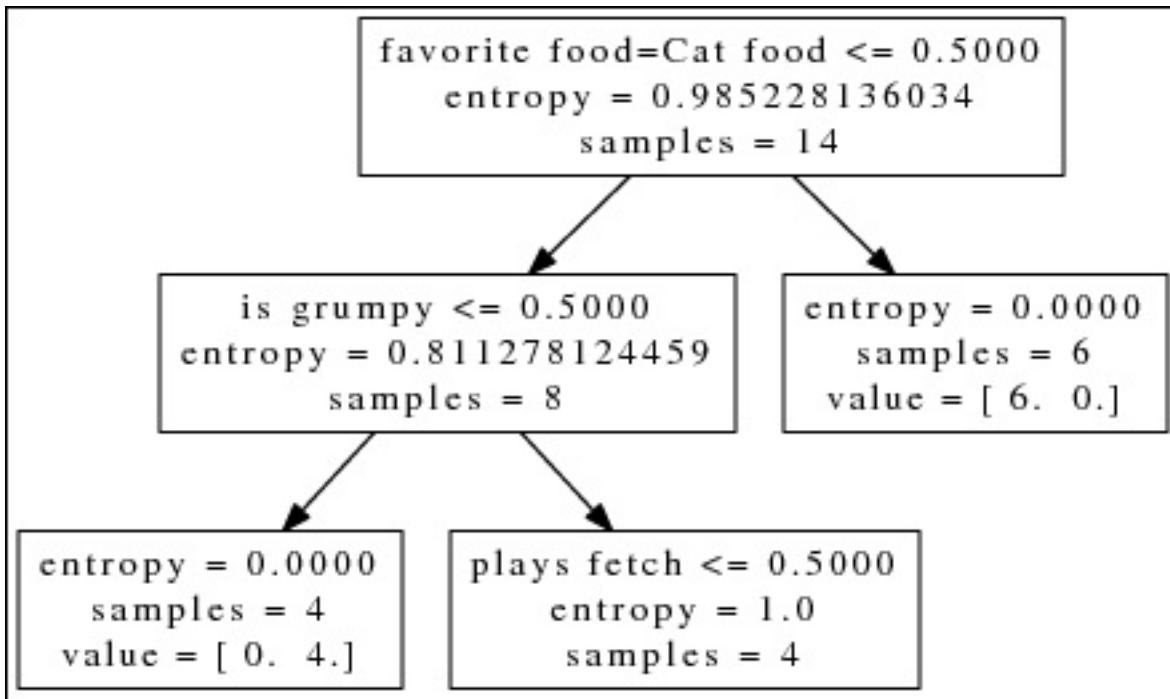
bacon				$* 5/14 = 0.9371$	
-------	--	--	--	-------------------	--

Now let's add another node to the tree. One of the child nodes produced by the test is a leaf node that contains only cats. The other node still contains two cats and six dogs. We will add a test to this node. Which of the remaining explanatory variables reduces our uncertainty the most? The following table contains the information gains for all of the possible tests:

Test	Parent's entropy	Child's entropy	Child's entropy	Weighted average	IG
plays fetch?	0.8113	1	0	$1.0 * 4/8 + 0 * 4/8 = 0.5$	0.3113
is grumpy?	0.8113	0	1	$0.0 * 4/8 + 1 * 4/8 = 0.5$	0.3113
favorite food=dog food	0.8113	0.9710	0	$0.9710 * 5/8 + 0.0 * 3/8 = 0.6069$	0.2044
favorite food=bacon	0.8113	0	0.9710	$0.0 * 3/8 + 0.9710 * 5/8 = 0.6069$	0.2044

All of the tests produce subsets with 0 bits of entropy, but

the `is grumpy` and `plays fetch` tests produce the greatest information gain. ID3 breaks ties by selecting one of the best tests arbitrarily. We will select the `is grumpy` test, which splits its parent's eight instances into a leaf node containing four dogs and a node containing two cats and two dogs. The following is a diagram of the current tree:

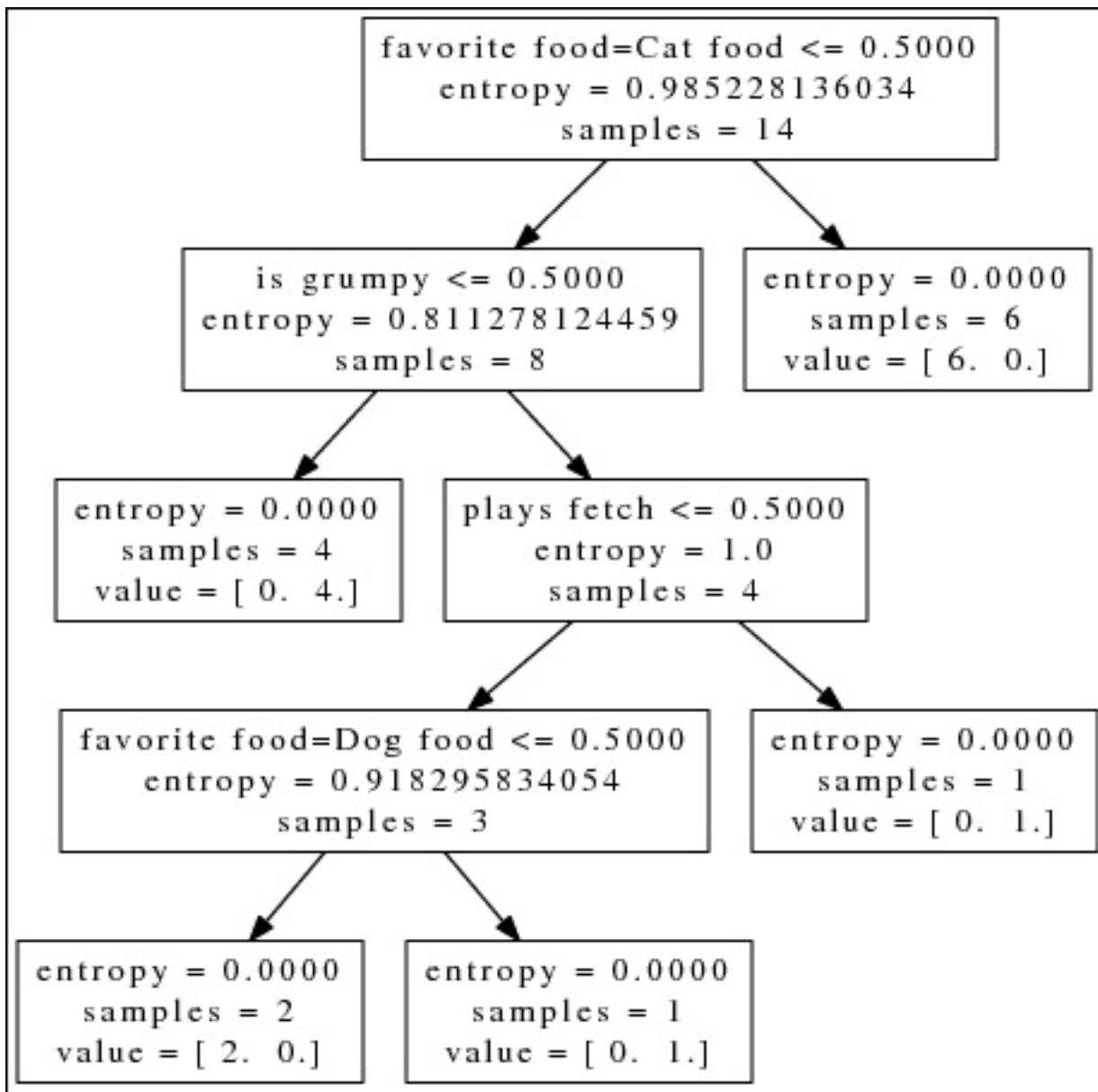


We will now select another explanatory variable to test the child node's four instances. The remaining tests, `favorite food=bacon`, `favorite food=dog food`, and `plays fetch`, all produce a leaf node containing one dog or cat and a node containing the remaining animals. The remaining tests produce equal information gains, as

shown in the following table:

Test	Parent's Entropy	Child Entropy	Child Entropy	Weighted Average	Information Gain
plays fetch?	1	0.9183	0	0.688725	0.311275
favorite food=dog food	1	0.9183	0	0.688725	0.311275
favorite food=bacon	1	0	0.9183	0.688725	0.311275

We will arbitrarily select the `plays fetch` test to produce a leaf node containing one dog and a node containing two cats and a dog. Two explanatory variables remain; we can test for animals that like bacon, or we can test for animals that like dog food. Both of the tests will produce the same subsets and create a leaf node containing one dog and a leaf node containing two cats. We will arbitrarily choose to test for animals that like dog food. The following is a diagram of the completed decision tree:



Let's classify some animals from the following test data:

Testing instance	Plays fetch	Is grumpy	Favorite food	Species

	Yes	No	Bacon	Dog
1	Yes	No	Bacon	Dog
2	Yes	Yes	Dog Food	Dog
3	No	Yes	Dog Food	Cat
4	No	Yes	Bacon	Cat
5	No	No	Cat food	Cat

Let's classify the first animal, which likes to plays fetch, is infrequently grumpy, and loves bacon. We will follow the edge to the root node's left child since the animal's favorite food is not cat food. The animal is not grumpy, so we will follow the edge to the second-level node's left child. This is a leaf node containing only dogs; we have correctly classified this instance. To classify the third test instance as a cat, we follow the edge to the root node's left child, follow the edge to the second-level node's right child, follow the edge to the third-level node's left child, and finally follow the edge to the fourth-level node's right child.

Congratulations! You've constructed a decision tree using the ID3 algorithm. Other algorithms can be used to train decision trees. **C4.5** is a modified version of ID3 that can

be used with continuous explanatory variables and can accommodate missing values for features. C4.5 also can **prune** trees. Pruning reduces the size of a tree by replacing branches that classify few instances with leaf nodes. Used by scikit-learn's implementation of decision trees, **CART** is another learning algorithm that supports pruning.

Gini impurity

In the previous section, we built a decision tree by creating nodes that produced the greatest information gain. Another common heuristic for learning decision trees is **Gini impurity**, which measures the proportions of classes in a set. Gini impurity is given by the following equation, where J is the number of classes, t is the subset of instances for the node, and $P(i|t)$ is the probability of selecting an element of class i from the node's subset:

$$Gini(t) = 1 - \sum_{i=1}^J P(i|t)^2$$

Intuitively, Gini impurity is zero when all of the elements of the set are the same class, as the probability of selecting an element of that class is equal to one. Like entropy, Gini impurity is greatest when each class has an equal probability of being selected. The maximum value of Gini impurity depends on the number of possible classes, and it is given by the following equation:

$$Gini_{\max} = 1 - \frac{1}{n}$$

Our problem has two classes, so the maximum value of the Gini impurity measure will be equal to one half. scikit-learn supports learning decision trees using both information gain and Gini impurity. There are no firm rules to help you decide when to use one criterion or the other; in practice, they often produce similar results. As with many decisions in machine learning, it is best to compare the performances of models trained using both options.

Decision trees with scikit-learn

Let's use decision trees to create software that can block banner ads on web pages. This program will predict whether each of the images on a web page is an advertisement or article content. Images that are classified as being advertisements could then be hidden using Cascading Style Sheets. We will train a decision tree classifier using the *Internet Advertisements Data Set* from [http://archive.ics.uci.edu/ml/datasets/Internet+Advertiseme](http://archive.ics.uci.edu/ml/datasets/Internet+Advertisements) which contains data for 3,279 images. The proportions of the classes are skewed; 459 of the images are advertisements and 2,820 are content. Decision tree learning algorithms can produce biased trees from data with unbalanced class proportions; we will evaluate a model on the unaltered data set before deciding if it is worth balancing the training data by over- or under-sampling instances. The explanatory variables are the dimensions of the image, words from the containing page's URL, words from the image's URL, the image's alt text, the image's anchor text, and a window of words surrounding the image tag. The response variable is the image's class. The explanatory variables have already been transformed into feature representations. The first

three features are real numbers that encode the width, height, and aspect ratio of the images. The remaining features encode binary term frequencies for the text variables. In the following sample, we will grid search for the hyperparameter values that produce the decision tree with the greatest accuracy, and then evaluate the tree's performance on a test set:

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.cross_validation import
train_test_split
from sklearn.metrics import
classification_report
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV
```

First we read the .csv file using pandas. The .csv does not have a header row, so we split the last column containing the response variable's values from the features using its index:

```
if __name__ == '__main__':
    df = pd.read_csv('data/ad.data',
header=None)
    explanatory_variable_columns =
set(df.columns.values)
    response_variable_column =
df[len(df.columns.values)-1]
    # The last column describes the targets
```

```
explanatory_variable_columns.remove(len(df.columns.values)-1)

y = [1 if e == 'ad.' else 0 for e in
response_variable_column]
x = df[list(explanatory_variable_columns)]
```

We encoded the advertisements as the positive class and the content as the negative class. More than one quarter of the instances are missing at least one of the values for the image's dimensions. These missing values are marked by whitespace and a question mark. We replaced the missing values with negative one, but we could have imputed the missing values; for instance, we could have replaced the missing height values with the average height value:

```
x.replace(to_replace=' *\\?', value=-1,
regex=True, inplace=True)
```

We then split the data into training and test sets:

```
x_train, x_test, y_train, y_test =
train_test_split(x, y)
```

We created a pipeline and an instance of `DecisionTreeClassifier`. Then, we set the `criterion` keyword argument to `entropy` to build the tree using the information gain heuristic:

```
pipeline = Pipeline([
```

```
('clf',
DecisionTreeClassifier(criterion='entropy'))
])
```

Next, we specified the hyperparameter space for the grid search:

```
parameters = {
    'clf__max_depth': (150, 155, 160),
    'clf__min_samples_split': (1, 2, 3),
    'clf__min_samples_leaf': (1, 2, 3)
}
```

We set `GridSearchCV()` to maximize the model's F1 score:

```
grid_search = GridSearchCV(pipeline,
parameters, n_jobs=-1, verbose=1, scoring='f1')
grid_search.fit(X_train, y_train)
print 'Best score: %0.3f' %
grid_search.best_score_
print 'Best parameters set:'
best_parameters =
grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print '\t%s: %r' % (param_name,
best_parameters[param_name])

predictions = grid_search.predict(X_test)
print classification_report(y_test,
predictions)
```

Fitting 3 folds for each of 27 candidates,

```

totalling 81 fits
[Parallel(n_jobs=-1)]: Done    1 jobs          |
elapsed: 1.7s
[Parallel(n_jobs=-1)]: Done   50 jobs          |
elapsed: 15.0s
[Parallel(n_jobs=-1)]: Done   71 out of 81 |
elapsed: 20.7s remaining: 2.9s
[Parallel(n_jobs=-1)]: Done   81 out of 81 |
elapsed: 23.3s finished
Best score: 0.878
Best parameters set:
      clf__max_depth: 155
      clf__min_samples_leaf: 2
      clf__min_samples_split: 1
                  precision    recall    f1-score
support
           0        0.97      0.99      0.98
710
           1        0.92      0.81      0.86
110
avg / total        0.96      0.96      0.96
820

```

The classifier detected more than 80 percent of the ads in the test set, and approximately 92 percent of the images that it predicted were ads were truly ads. Overall, the performance is promising; in following sections, we will try to modify our model to improve its performance.

Tree ensembles

Ensemble learning methods combine a set of models to produce an estimator that has better predictive performance than its individual components. A **random forest** is a collection of decision trees that have been trained on randomly selected subsets of the training instances and explanatory variables. Random forests usually make predictions by returning the mode or mean of the predictions of their constituent trees; scikit-learn's implementations return the mean of the trees' predictions. Random forests are less prone to overfitting than decision trees because no single tree can learn from all of the instances and explanatory variables; no single tree can memorize all of the noise in the representation.

Let's update our ad blocker's classifier to use a random forest. It is simple to replace the `DecisionTreeClassifier` using scikit-learn's API; we simply replace the object with an instance of `RandomForestClassifier`. Like the previous example, we will grid search to find the values of the hyperparameters that produce the random forest with the best predictive performance.

First, import the `RandomForestClassifier` class from the

ensemble module:

```
from sklearn.ensemble import  
RandomForestClassifier
```

Next, replace the DecisionTreeClassifier in the pipeline with an instance of RandomForestClassifier and update the hyperparameter space:

```
pipeline = Pipeline([  
    ('clf',  
     RandomForestClassifier(criterion='entropy'))  
])  
parameters = {  
    'clf__n_estimators': (5, 10, 20, 50),  
    'clf__max_depth': (50, 150, 250),  
    'clf__min_samples_split': (1, 2, 3),  
    'clf__min_samples_leaf': (1, 2, 3)  
}
```

The output is as follows:

```
Fitting 3 folds for each of 108 candidates,  
totalling 324 fits  
[Parallel(n_jobs=-1)]: Done 1 jobs |  
elapsed: 1.1s  
[Parallel(n_jobs=-1)]: Done 50 jobs |  
elapsed: 17.4s  
[Parallel(n_jobs=-1)]: Done 200 jobs |  
elapsed: 1.0min  
[Parallel(n_jobs=-1)]: Done 324 out of 324 |  
elapsed: 1.6min finished  
Best score: 0.936
```

Best parameters set:

clf_max_depth: 250
clf_min_samples_leaf: 1
clf_min_samples_split: 3
clf_n_estimators: 20

		precision	recall	f1-score
	support			
705	0	0.97	1.00	0.98
115	1	0.97	0.83	0.90
820	avg / total	0.97	0.97	0.97

Replacing the single decision tree with a random forest resulted in a significant reduction of the error rate; the random forest improves the precision and recall for ads to 0.97 and 0.83.

The advantages and disadvantages of decision trees

The compromises associated with using decision trees are different than those of the other models we discussed. Decision trees are easy to use. Unlike many learning algorithms, decision trees do not require the data to have zero mean and unit variance. While decision trees can tolerate missing values for explanatory variables, scikit-learn's current implementation cannot. Decision trees can even learn to ignore explanatory variables that are not relevant to the task.

Small decision trees can be easy to interpret and visualize with the `export_graphviz` function from scikit-learn's `tree` module. The branches of a decision tree are conjunctions of logical predicates, and they are easily visualized as flowcharts. Decision trees support multioutput tasks, and a single decision tree can be used for multiclass classification without employing a strategy like one-versus-all.

Like the other models we discussed, decision trees are **eager learners**. Eager learners must build an input-independent model from the training data before they can be used to estimate the values of test instances, but can

predict relatively quickly once the model has been built. In contrast, **lazy learners** such as the k-nearest neighbors algorithm defer all generalization until they must make a prediction. Lazy learners do not spend time training, but often predict slowly compared to eager learners.

Decision trees are more prone to overfitting than many of the models we discussed, as their learning algorithms can produce large, complicated decision trees that perfectly model every training instance but fail to generalize the real relationship. Several techniques can mitigate overfitting in decision trees. **Pruning** is a common strategy that removes some of the tallest nodes and leaves of a decision tree, but it is not currently implemented in scikit-learn. However, similar effects can be achieved by setting a maximum depth for the tree or by creating child nodes only when the number of training instances they will contain exceeds a threshold. The

`DecisionTreeClassifier` and `DecisionTreeRegressor` classes provide keyword arguments to set these constraints. Creating a random forest can also reduce over-fitting.

Efficient decision tree learning algorithms like ID3 are **greedy**. They learn efficiently by making locally optimal decisions, but are not guaranteed to produce the globally optimal tree. ID3 constructs a tree by selecting a sequence

of explanatory variables to test. Each explanatory variable is selected because it reduces the uncertainty in the node more than the other variables. It is possible, however, that locally suboptimal tests are required in order to find the globally optimal tree.

In our toy examples, the size of the tree did not matter since we retained all of nodes. In a real application, however, the tree's growth could be limited by pruning or similar mechanisms. Pruning trees with different shapes can produce trees with different performances. In practice, locally optimal decisions that are guided by the information gain or Gini impurity heuristics often result in an acceptable decision trees.

Summary

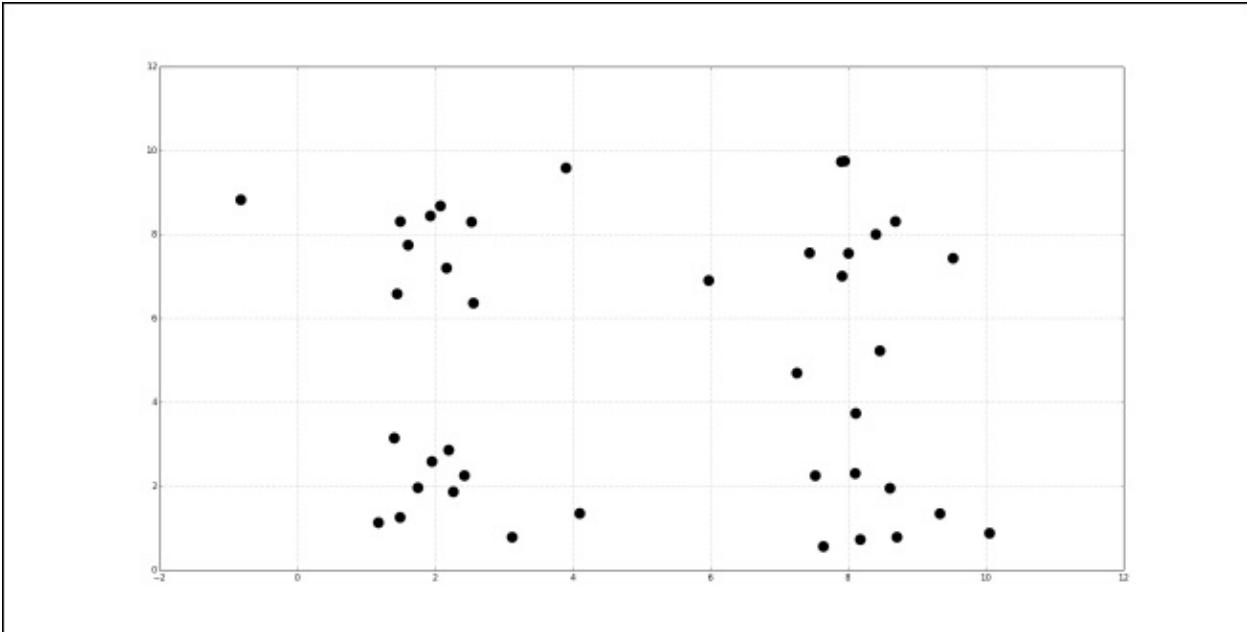
In this chapter we learned about simple nonlinear models for classification and regression called decision trees. Like the parlor game Twenty Questions, decision trees are composed of sequences of questions that examine a test instance. The branches of a decision tree terminate in leaves that specify the predicted value of the response variable. We discussed how to train decision trees using the ID3 algorithm, which recursively splits the training instances into subsets that reduce our uncertainty about the value of the response variable. We also discussed ensemble learning methods, which combine the predictions from a set of models to produce an estimator with better predictive performance. Finally, we used random forests to predict whether or not an image on a web page is a banner advertisement. In the next chapter, we will introduce our first unsupervised learning task: clustering.

Chapter 6. Clustering with K-Means

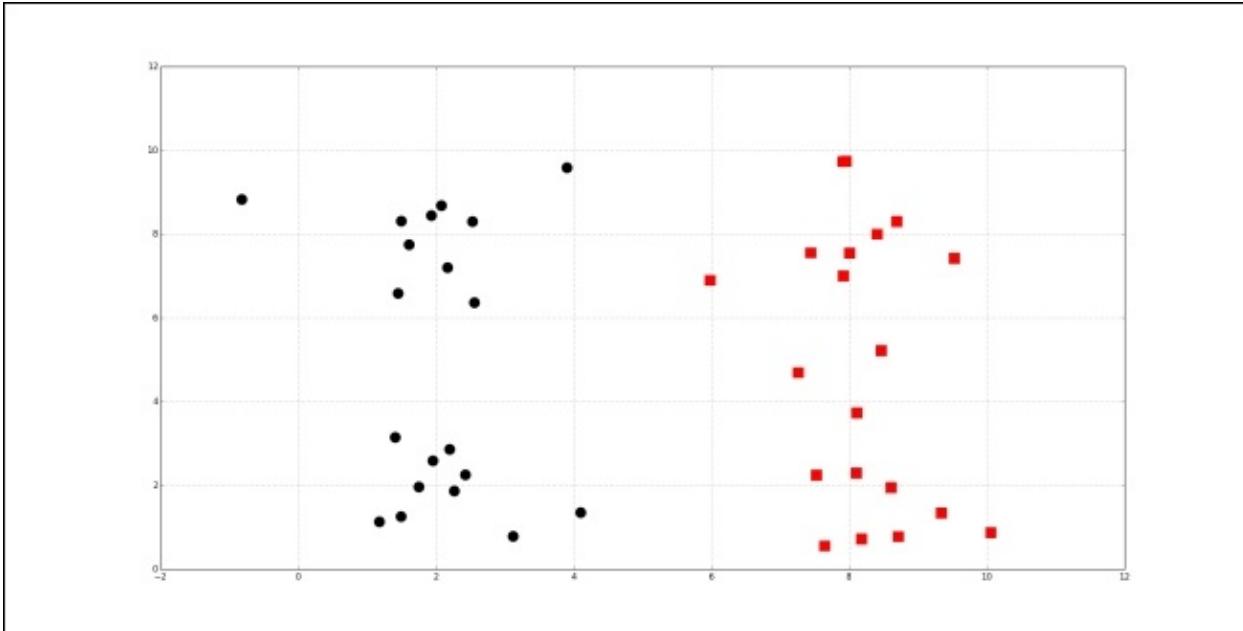
In the previous chapters we discussed supervised learning tasks; we examined algorithms for regression and classification that learned from labeled training data. In this chapter we will discuss an unsupervised learning task called clustering. Clustering is used to find groups of similar observations within a set of unlabeled data. We will discuss the K-Means clustering algorithm, apply it to an image compression problem, and learn to measure its performance. Finally, we will work through a semi-supervised learning problem that combines clustering with classification.

Recall from [Chapter 1](#), *The Fundamentals of Machine Learning*, that the goal of unsupervised learning is to discover hidden structure or patterns in unlabeled training data. **Clustering**, or **cluster analysis**, is the task of grouping observations such that members of the same group, or cluster, are more similar to each other by a given metric than they are to the members of the other clusters. As with supervised learning, we will represent an observation as an n -dimensional vector. For example, assume that your training data consists of the samples

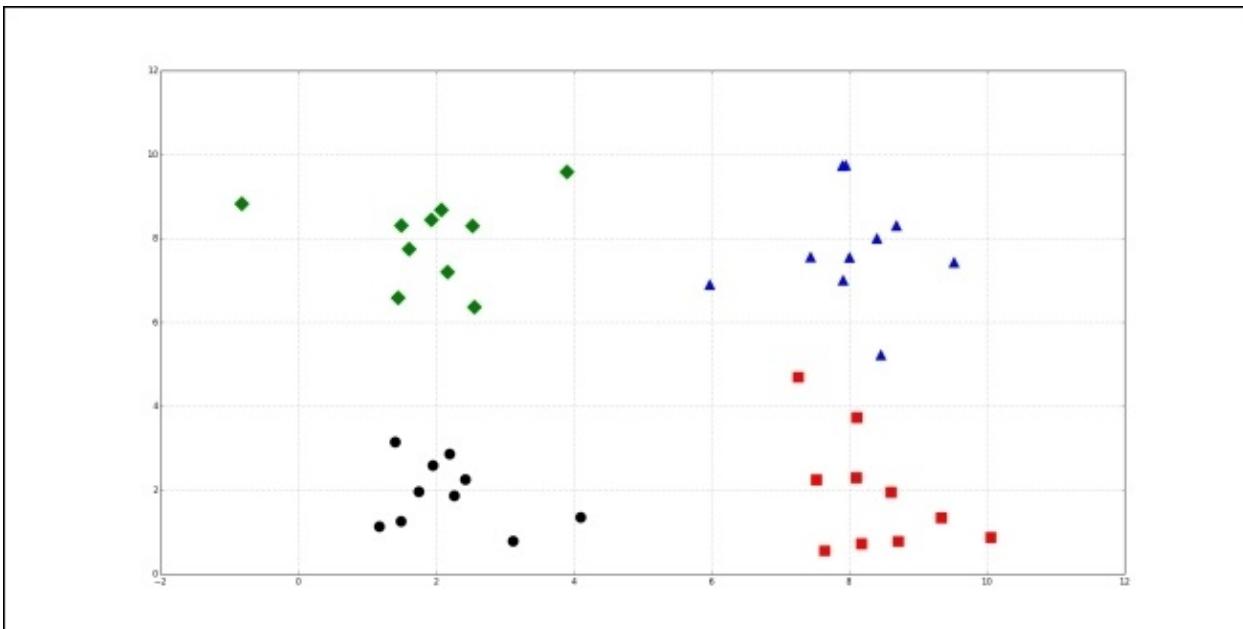
plotted in the following figure:



Clustering might reveal the following two groups, indicated by squares and circles:



Clustering could also reveal the following four groups:



Clustering is commonly used to explore a dataset. Social networks can be clustered to identify communities and to suggest missing connections between people. In biology, clustering is used to find groups of genes with similar expression patterns. Recommendation systems sometimes employ clustering to identify products or media that might appeal to a user. In marketing, clustering is used to find segments of similar consumers. In the following sections, we will work through an example of using the K-Means algorithm to cluster a dataset.

Clustering with the K-Means algorithm

The K-Means algorithm is a clustering method that is popular because of its speed and scalability. K-Means is an iterative process of moving the centers of the clusters, or the **centroids**, to the mean position of their constituent points, and re-assigning instances to their closest clusters. The titular K is a hyperparameter that specifies the number of clusters that should be created; K-Means automatically assigns observations to clusters but cannot determine the appropriate number of clusters. K must be a positive integer that is less than the number of instances in the training set. Sometimes, the number of clusters is specified by the clustering problem's context. For example, a company that manufactures shoes might know that it is able to support manufacturing three new models. To understand what groups of customers to target with each model, it surveys customers and creates three clusters from the results. That is, the value of K was specified by the problem's context. Other problems may not require a specific number of clusters, and the optimal number of clusters may be ambiguous. We will discuss a heuristic to estimate the optimal number of clusters called the elbow method later in this chapter.

The parameters of K-Means are the positions of the clusters' centroids and the observations that are assigned to each cluster. Like generalized linear models and decision trees, the optimal values of K-Means' parameters are found by minimizing a cost function. The cost function for K-Means is given by the following equation:

$$J = \sum_{k=1}^K \sum_{i \in C_k} \|x_i - \mu_k\|^2$$

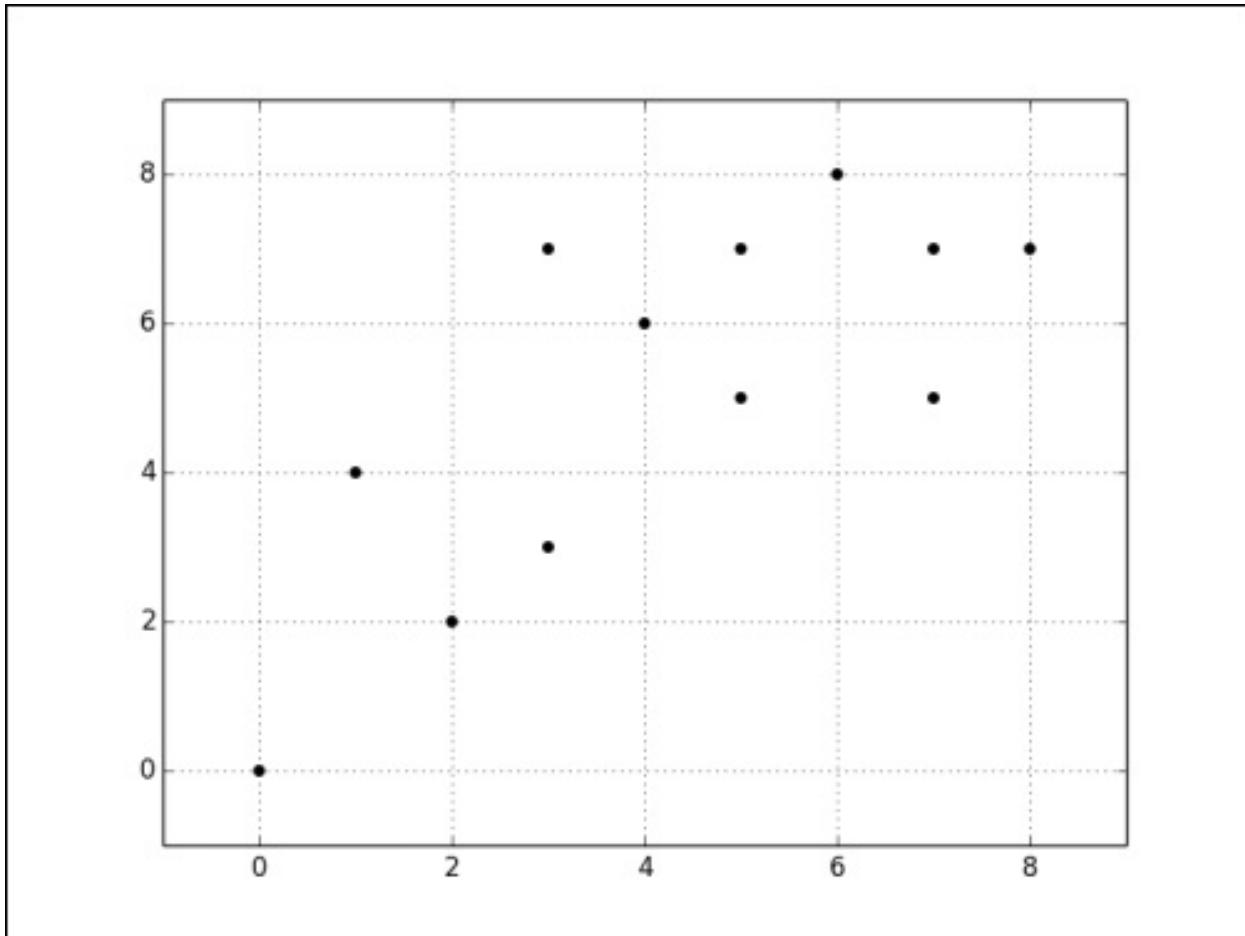
In the preceding equation, μ_k is the centroid for the cluster k . The cost function sums the distortions of the clusters. Each cluster's distortion is equal to the sum of the squared distances between its centroid and its constituent instances. The distortion is small for compact clusters and large for clusters that contain scattered instances. The parameters that minimize the cost function are learned through an iterative process of assigning observations to clusters and then moving the clusters. First, the clusters' centroids are initialized to random positions. In practice, setting the centroids' positions equal to the positions of randomly selected observations yields the best results. During each iteration, K-Means assigns observations to the cluster that they are closest to, and then moves the centroids to their assigned observations' mean location.

Let's work through an example by hand using the training data shown in the following table:

Instance	X0	X1
1	7	5
2	5	7
3	7	7
4	3	3
5	4	6
6	1	4
7	0	0
8	2	2
9	8	7
10	6	8
11	5	5

12	3	7
----	---	---

There are two explanatory variables and each instance has two features. The instances are plotted in the following figure:



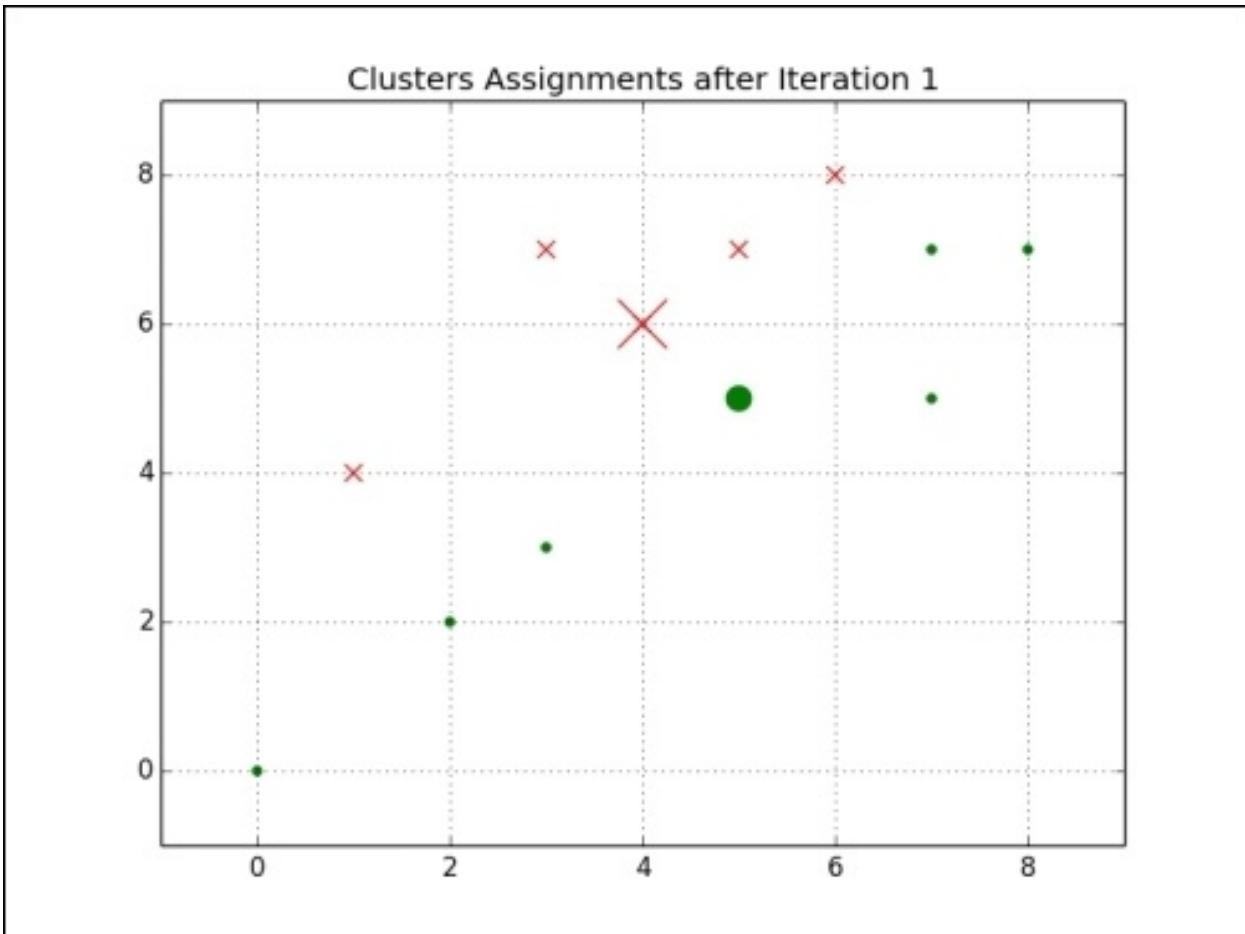
Assume that K-Means initializes the centroid for the first cluster to the fifth instance and the centroid for the second

cluster to the eleventh instance. For each instance, we will calculate its distance to both centroids, and assign it to the cluster with the closest centroid. The initial assignments are shown in the **Cluster** column of the following table:

Instance	X0	X1	C1 distance	C2 distance	Last cluster	Cluster	Changed?
1	7	5	3.16228	2	None	C2	Yes
2	5	7	1.41421	2	None	C1	Yes
3	7	7	3.16228	2.82843	None	C2	Yes
4	3	3	3.16228	2.82843	None	C2	Yes
5	4	6	0	1.41421	None	C1	Yes
6	1	4	3.60555	4.12311	None	C1	Yes
7	0	0	7.21110	7.07107	None	C2	Yes
8	2	2	4.47214	4.24264	None	C2	Yes
9	8	7	4.12311	3.60555	None	C2	Yes

10	6	8	2.82843	3.16228	None	C1	Yes
11	5	5	1.41421	0	None	C2	Yes
12	3	7	1.41421	2.82843	None	C1	Yes
C1 centroid	4	6					
C2 centroid	5	5					

The plotted centroids and the initial cluster assignments are shown in the following graph. Instances assigned to the first cluster are marked with Xs, and instances assigned to the second cluster are marked with dots. The markers for the centroids are larger than the markers for the instances.



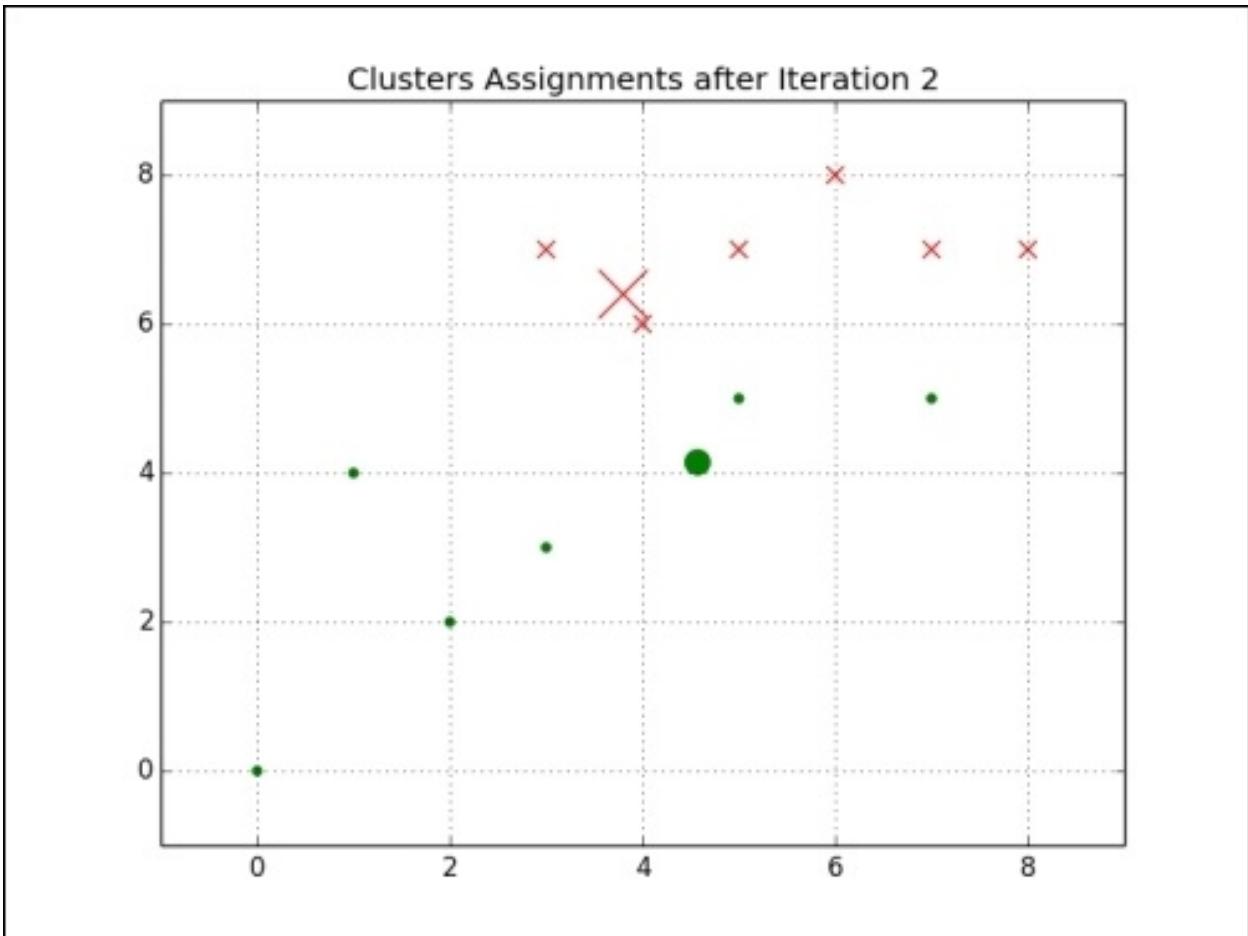
Now we will move both centroids to the means of their constituent instances, recalculate the distances of the training instances to the centroids, and reassign the instances to the closest centroids:

Instance	X0	X1	C1 distance	C2 distance	Last Cluster	New Cluster	Chang
1	7	5	3.492850	2.575394	C2	C2	No

2	5	7	1.341641	2.889107	C1	C1	No
3	7	7	3.255764	3.749830	C2	C1	Yes
4	3	3	3.492850	1.943067	C2	C2	No
5	4	6	0.447214	1.943067	C1	C1	No
6	1	4	3.687818	3.574285	C1	C2	Yes
7	0	0	7.443118	6.169378	C2	C2	No
8	2	2	4.753946	3.347250	C2	C2	No
9	8	7	4.242641	4.463000	C2	C1	Yes
10	6	8	2.720294	4.113194	C1	C1	No
11	5	5	1.843909	0.958315	C2	C2	No
12	3	7	1	3.260775	C1	C1	No
C1 centroid	3.8	6.4					

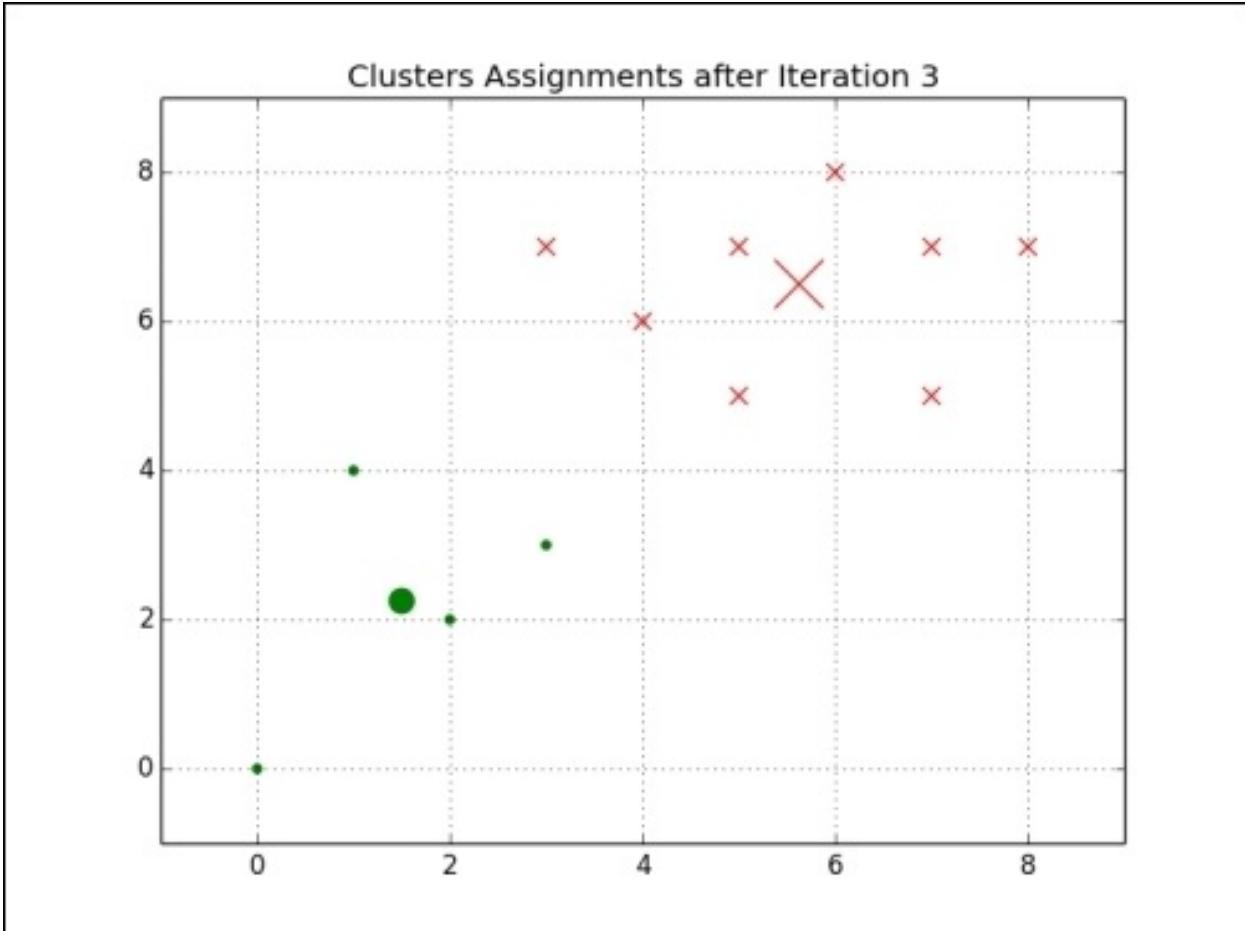
C2 centroid		4.571429		4.142857					
----------------	--	----------	--	----------	--	--	--	--	--

The new clusters are plotted in the following graph. Note that the centroids are diverging and several instances have changed their assignments:



Now, we will move the centroids to the means of their constituents' locations again and reassign the instances to

their nearest centroids. The centroids continue to diverge, as shown in the following figure:

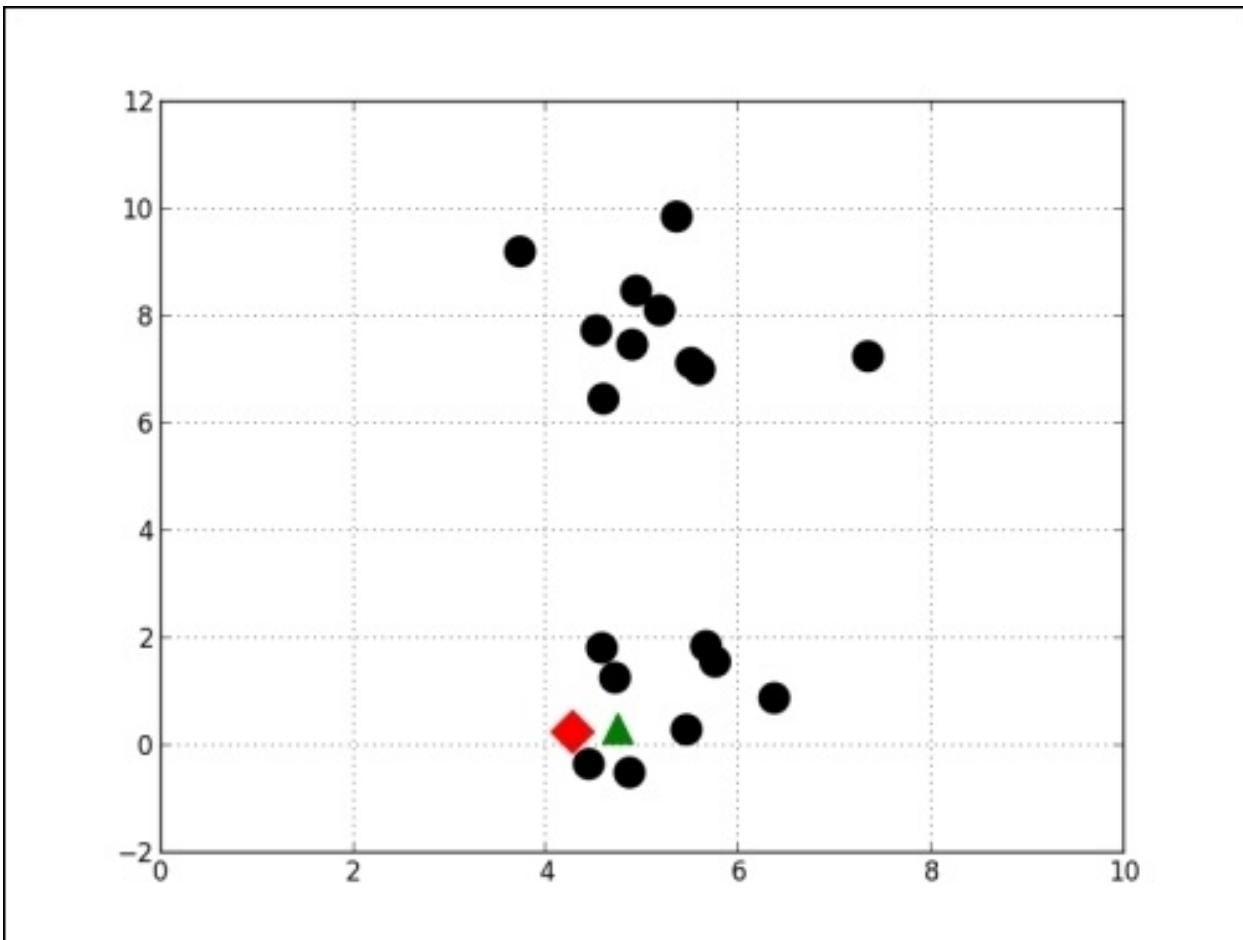


None of the instances' centroid assignments will change in the next iteration; K-Means will continue iterating until some stopping criteria is satisfied. Usually, this criterion is either a threshold for the difference between the values of the cost function for subsequent iterations, or a threshold for the change in the positions of the centroids

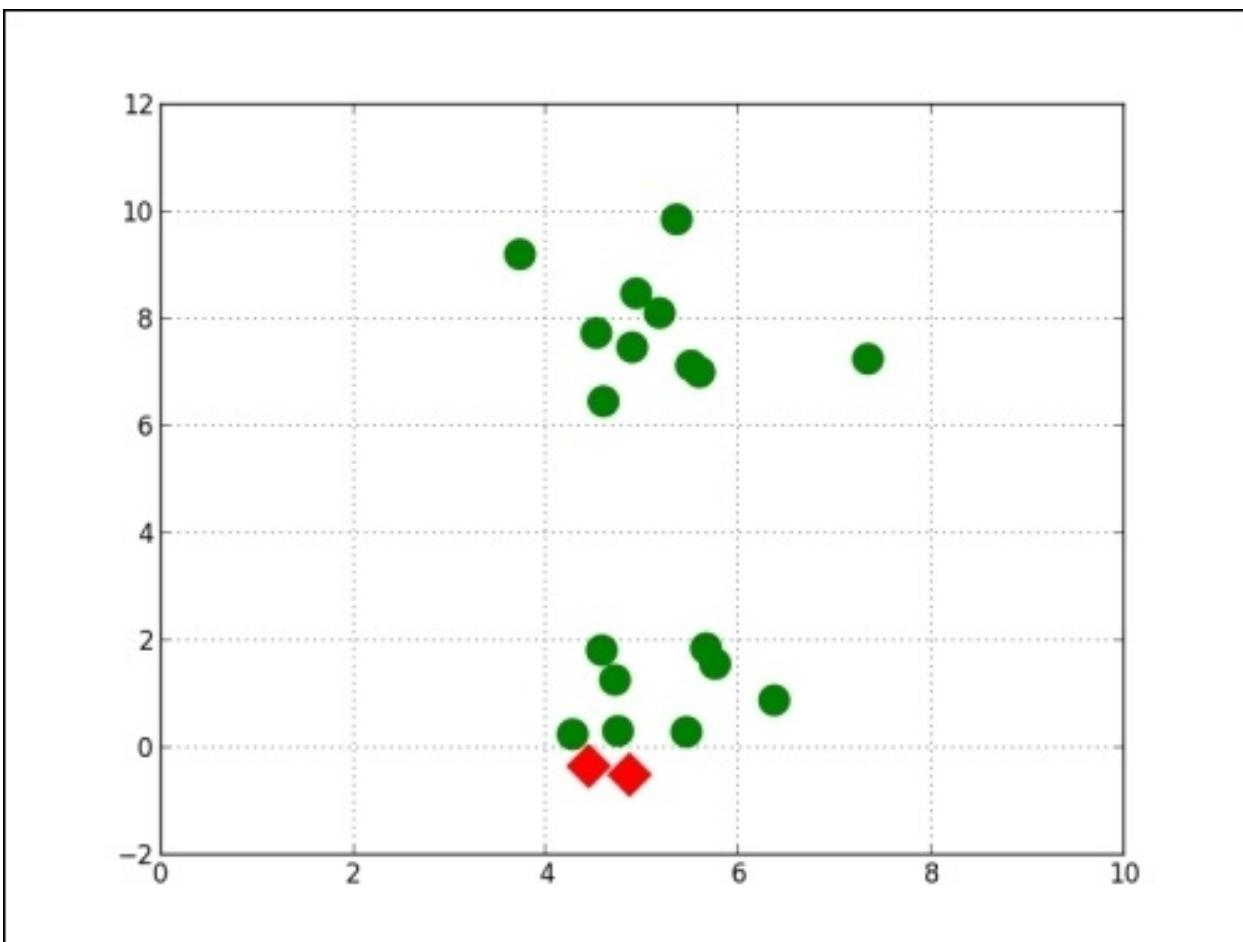
between subsequent iterations. If these stopping criteria are small enough, K-Means will converge on an optimum. This optimum will not necessarily be the global optimum.

Local optima

Recall that K-Means initially sets the positions of the clusters' centroids to the positions of randomly selected observations. Sometimes, the random initialization is unlucky and the centroids are set to positions that cause K-Means to converge to a local optimum. For example, assume that K-Means randomly initializes two cluster centroids to the following positions:

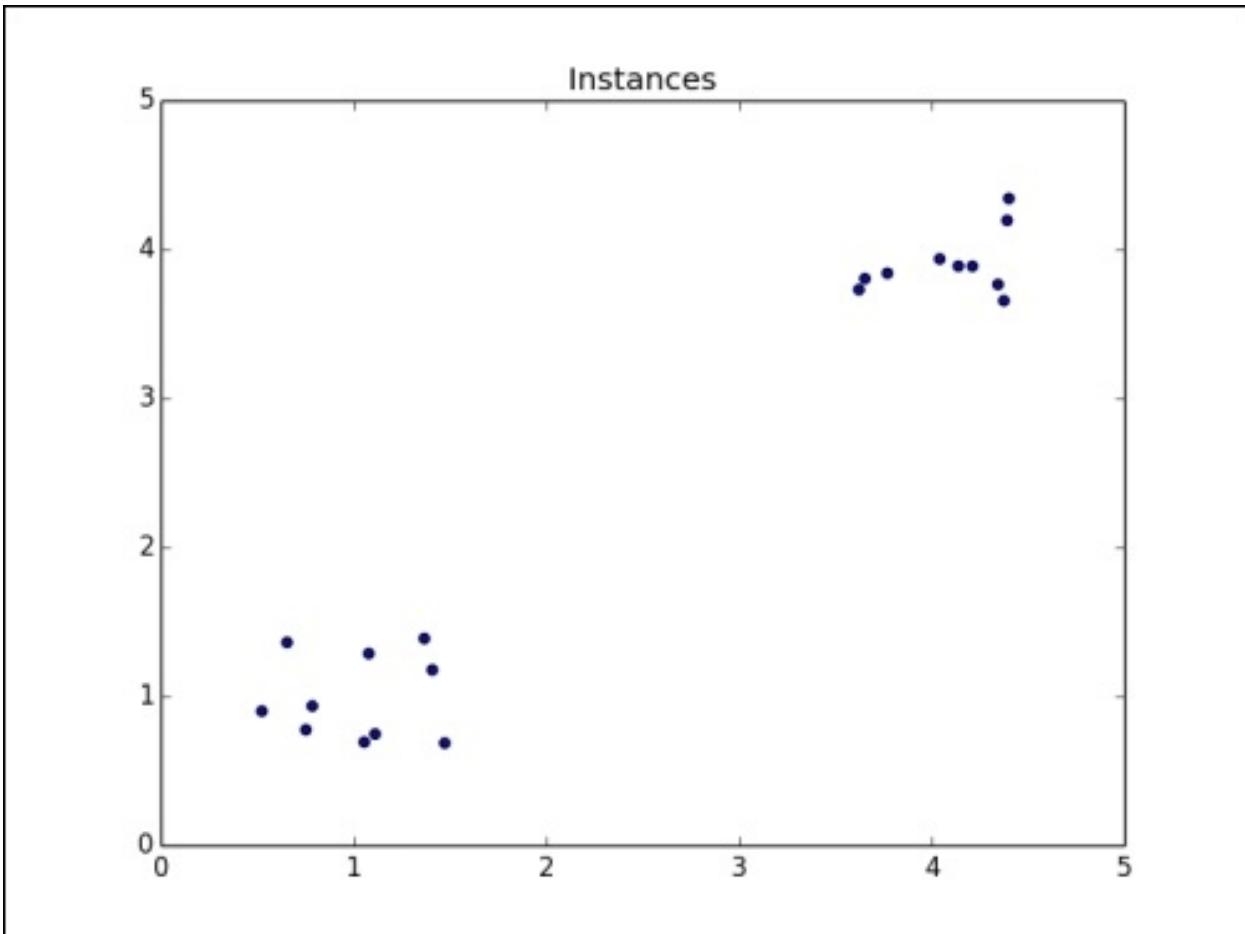


K-Means will eventually converge on a local optimum like that shown in the following figure. These clusters may be informative, but it is more likely that the top and bottom groups of observations are more informative clusters. To avoid local optima, K-Means is often repeated dozens or even hundreds of times. In each iteration, it is randomly initialized to different starting cluster positions. The initialization that minimizes the cost function best is selected.



The elbow method

If K is not specified by the problem's context, the optimal number of clusters can be estimated using a technique called the **elbow method**. The elbow method plots the value of the cost function produced by different values of K . As K increases, the average distortion will decrease; each cluster will have fewer constituent instances, and the instances will be closer to their respective centroids. However, the improvements to the average distortion will decline as K increases. The value of K at which the improvement to the distortion declines the most is called the elbow. Let's use the elbow method to choose the number of clusters for a dataset. The following scatter plot visualizes a dataset with two obvious clusters:



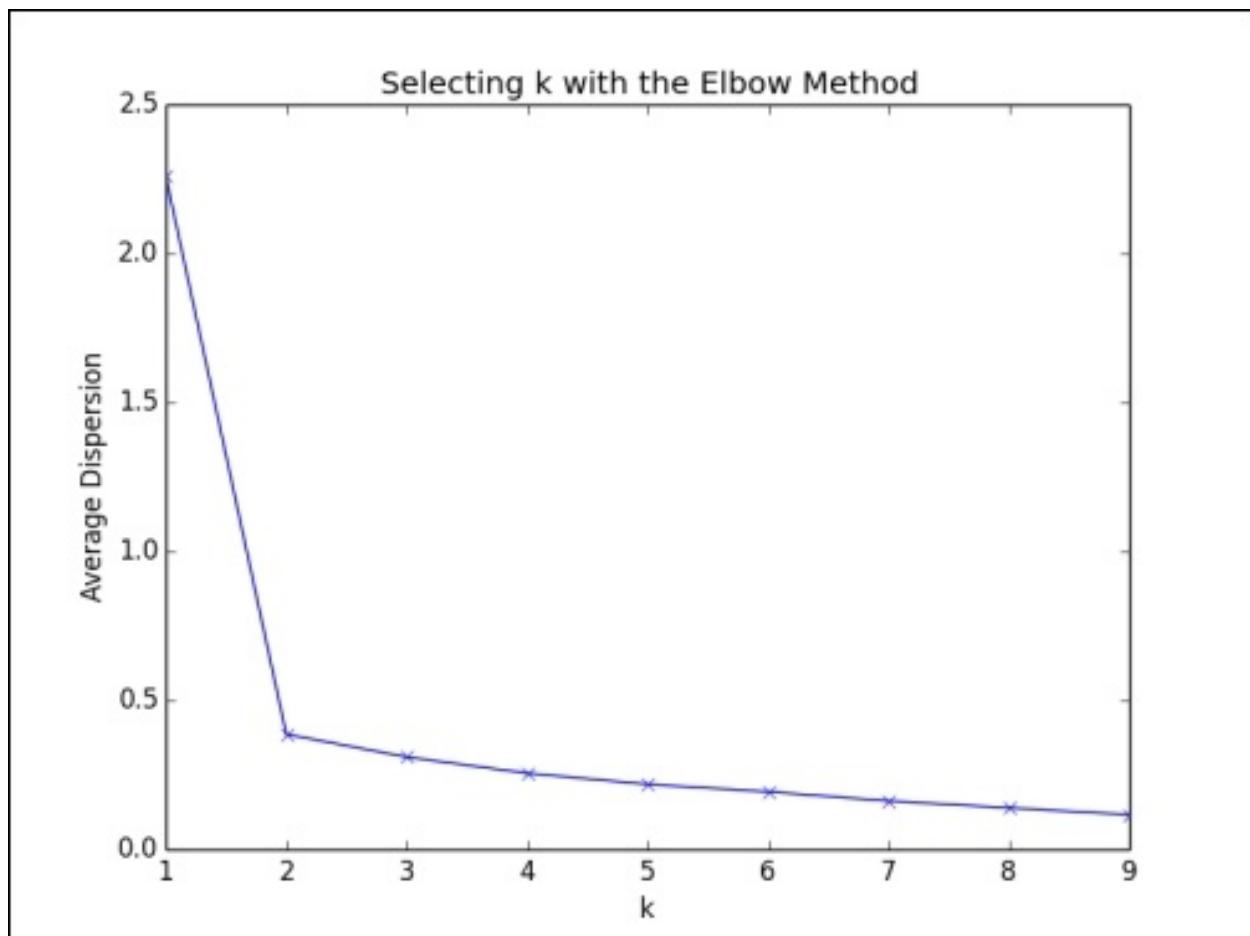
We will calculate and plot the mean distortion of the clusters for each value of K from 1 to 10 with the following code:

```
>>> import numpy as np  
>>> from sklearn.cluster import KMeans  
>>> from scipy.spatial.distance import cdist  
>>> import matplotlib.pyplot as plt  
  
>>> cluster1 = np.random.uniform(0.5, 1.5, (2,
```

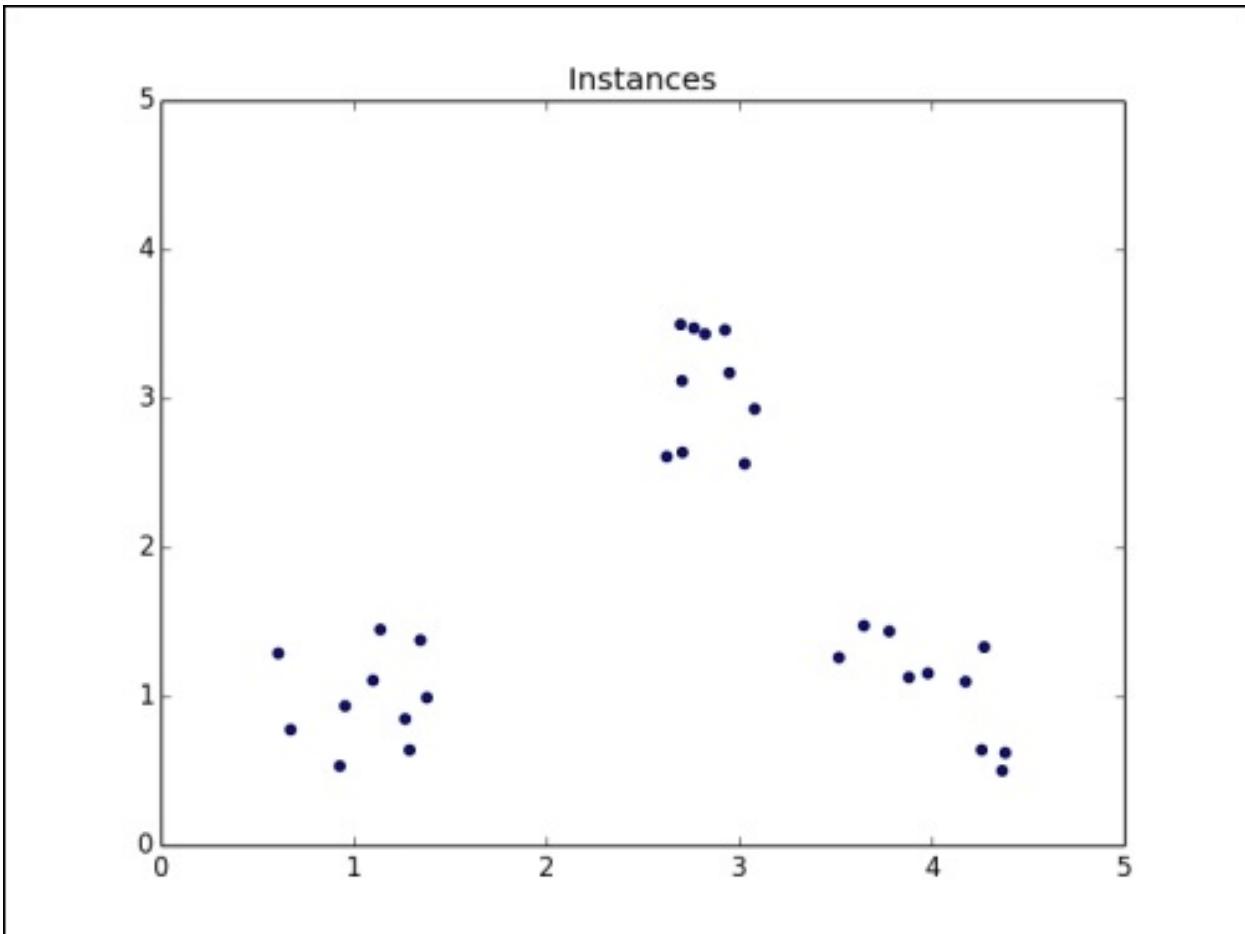
```
10))
>>> cluster2 = np.random.uniform(3.5, 4.5, (2,
10))
>>> X = np.hstack((cluster1, cluster2)).T
>>> X = np.vstack((x, y)).T

>>> K = range(1, 10)
>>> meandistortions = []
>>> for k in K:
>>>     kmeans = KMeans(n_clusters=k)
>>>     kmeans.fit(X)
>>>
meandistortions.append(sum(np.min(cdist(X,
kmeans.cluster_centers_, 'euclidean'), axis=1)) /
X.shape[0])

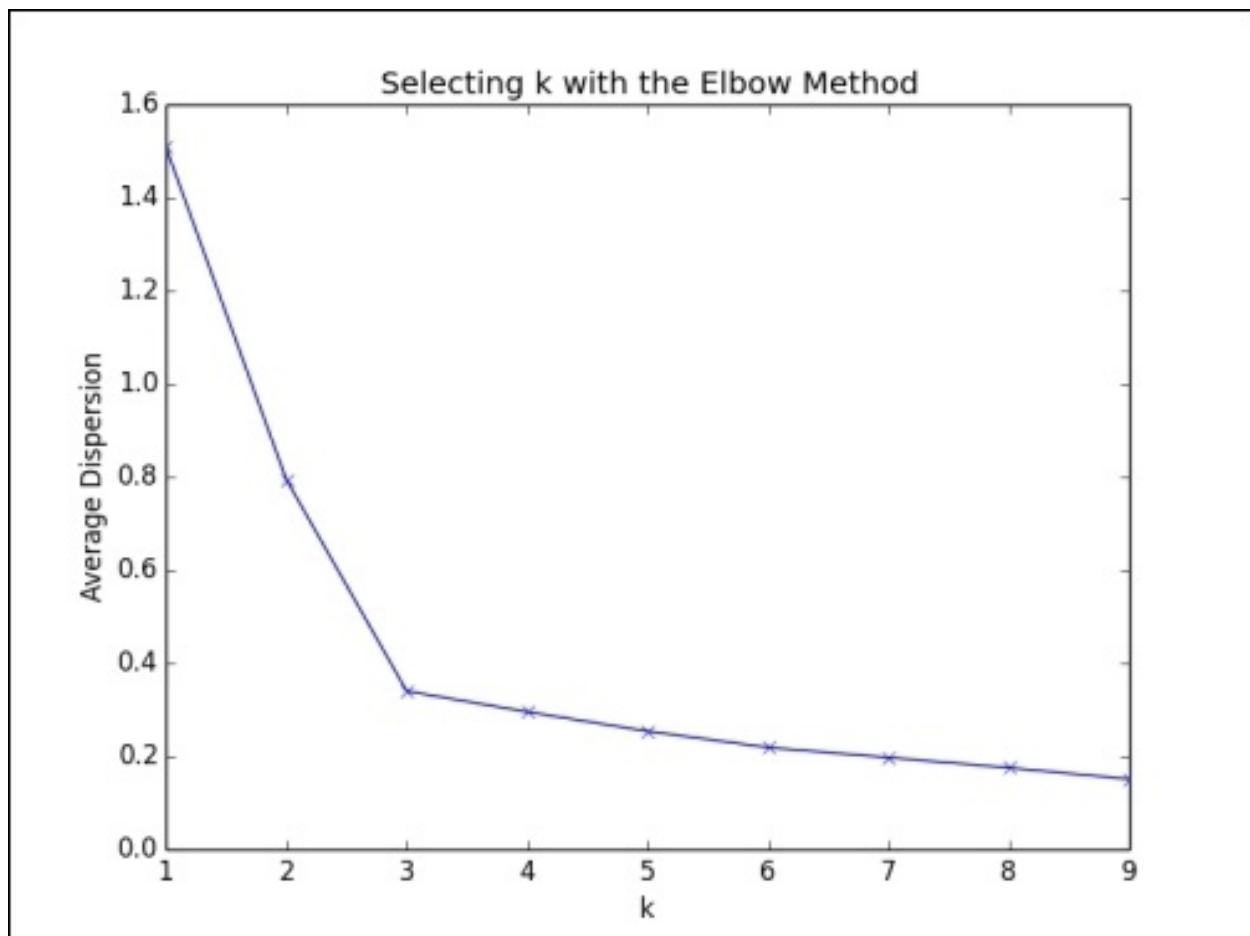
>>> plt.plot(K, meandistortions, 'bx-')
>>> plt.xlabel('k')
>>> plt.ylabel('Average distortion')
>>> plt.title('Selecting k with the Elbow
Method')
>>> plt.show()
```



The average distortion improves rapidly as we increase K from **1** to **2**. There is little improvement for values of K greater than 2. Now let's use the elbow method on the following dataset with three clusters:



The following figure shows the elbow plot for the dataset. From this, we can see that the rate of improvement to the average distortion declines the most when adding a fourth cluster, that is, the elbow method confirms that K should be set to three for this dataset.



Evaluating clusters

We defined machine learning as the design and study of systems that learn from experience to improve their performance of a task as measured by a given metric. K-Means is an unsupervised learning algorithm; there are no labels or ground truth to compare with the clusters.

However, we can still evaluate the performance of the algorithm using intrinsic measures. We have already discussed measuring the distortions of the clusters. In this section, we will discuss another performance measure for clustering called the **silhouette coefficient**. The silhouette coefficient is a measure of the compactness and separation of the clusters. It increases as the quality of the clusters increase; it is large for compact clusters that are far from each other and small for large, overlapping clusters. The silhouette coefficient is calculated per instance; for a set of instances, it is calculated as the mean of the individual samples' scores. The silhouette coefficient for an instance is calculated with the following equation:

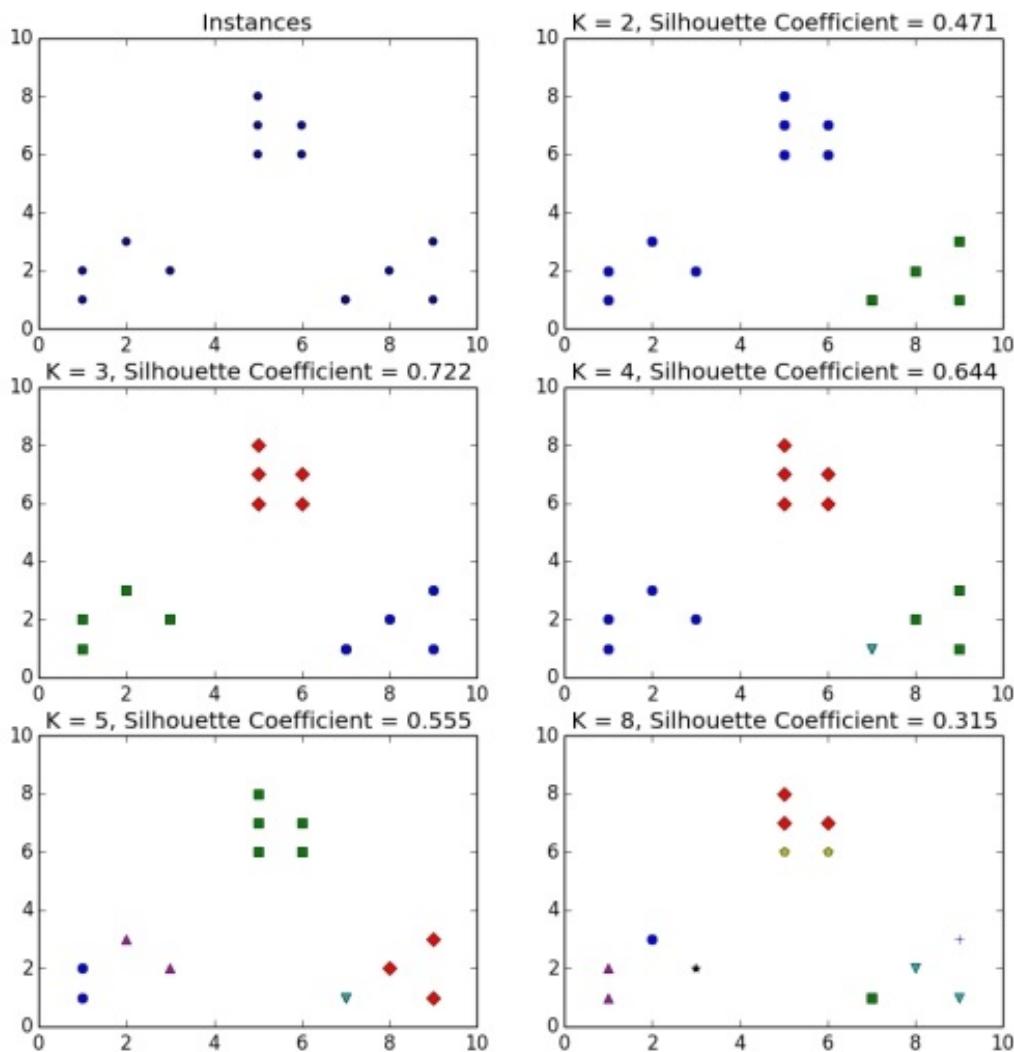
$$s = \frac{ba}{\max(a,b)}$$

a is the mean distance between the instances in the cluster. b is the mean distance between the instance and the instances in the next closest cluster. The following example runs K-Means four times to create two, three, four, and eight clusters from a toy dataset and calculates the silhouette coefficient for each run:

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> from sklearn import metrics
>>> import matplotlib.pyplot as plt
>>> plt.subplot(3, 2, 1)
>>> x1 = np.array([1, 2, 3, 1, 5, 6, 5, 5, 6, 7,
8, 9, 7, 9])
>>> x2 = np.array([1, 3, 2, 2, 8, 6, 7, 6, 7, 1,
2, 1, 1, 3])
>>> X = np.array(zip(x1, x2)).reshape(len(x1),
2)
>>> plt.xlim([0, 10])
>>> plt.ylim([0, 10])
>>> plt.title('Instances')
>>> plt.scatter(x1, x2)
>>> colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k',
'b']
>>> markers = ['o', 's', 'D', 'v', '^', 'p',
'*', '+']
>>> tests = [2, 3, 4, 5, 8]
>>> subplot_counter = 1
>>> for t in tests:
>>>     subplot_counter += 1
>>>     plt.subplot(3, 2, subplot_counter)
>>>     kmeans_model =
```

```
KMeans(n_clusters=t).fit(X)
>>>     for i, l in
enumerate(kmeans_model.labels_):
>>>         plt.plot(x1[i], x2[i],
color=colors[l], marker=markers[l], ls='None')
>>>         plt.xlim([0, 10])
>>>         plt.ylim([0, 10])
>>>         plt.title('K = %s, silhouette
coefficient = %.03f' % (
>>>             t, metrics.silhouette_score(X,
kmeans_model.labels_, metric='euclidean')) )
>>> plt.show()
```

This script produces the following figure:



The dataset contains three obvious clusters. Accordingly, the silhouette coefficient is greatest when K is equal to three. Setting K equal to eight produces clusters of instances that are as close to each other as they are to the

instances in some of the other clusters, and the silhouette coefficient of these clusters is smallest.

Image quantization

In the previous sections, we used clustering to explore the structure of a dataset. Now let's apply it to a different problem. Image quantization is a lossy compression method that replaces a range of similar colors in an image with a single color. Quantization reduces the size of the image file since fewer bits are required to represent the colors. In the following example, we will use clustering to discover a compressed palette for an image that contains its most important colors. We will then rebuild the image using the compressed palette. This example requires the `mahotas` image processing library, which can be installed using `pip install mahotas`:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.cluster import KMeans
>>> from sklearn.utils import shuffle
>>> import mahotas as mh
```

First we read and flatten the image:

```
>>> original_img =
np.array(mh.imread('img/tree.jpg'),
dtype=np.float64) / 255
>>> original_dimensions =
tuple(original_img.shape)
>>> width, height, depth =
```

```
tuple(original_img.shape)
>>> image_flattened = np.reshape(original_img,
(width * height, depth))
```

We then use K-Means to create 64 clusters from a sample of 1,000 randomly selected colors. Each of the clusters will be a color in the compressed palette. The code is as follows:

```
>>> image_array_sample =
shuffle(image_flattened, random_state=0) [:1000]
>>> estimator = KMeans(n_clusters=64,
random_state=0)
>>> estimator.fit(image_array_sample)
```

Next, we predict the cluster assignment for each of the pixels in the original image:

```
>>> cluster_assignments =
estimator.predict(image_flattened)
```

Finally, we create the compressed image from the compressed palette and cluster assignments:

```
>>> compressed_palette =
estimator.cluster_centers_
>>> compressed_img = np.zeros((width, height,
compressed_palette.shape[1]))
>>> label_idx = 0
>>> for i in range(width):
>>>     for j in range(height):
>>>         compressed_img[i][j] =
```

```
compressed_palette[cluster_assignments[label_idx]]  
>>> label_idx += 1  
>>> plt.subplot(122)  
>>> plt.title('Original Image')  
>>> plt.imshow(original_img)  
>>> plt.axis('off')  
>>> plt.subplot(121)  
>>> plt.title('Compressed Image')  
>>> plt.imshow(compressed_img)  
>>> plt.axis('off')  
>>> plt.show()
```

The original and compressed versions of the image are show in the following figure:



Clustering to learn features

In this example, we will combine clustering with classification in a semi-supervised learning problem. You will learn features by clustering unlabeled data and use the learned features to build a supervised classifier.

Suppose you own a cat and a dog. Suppose that you have purchased a smartphone, ostensibly to use to communicate with humans, but in practice just to use to photograph your cat and dog. Your photographs are awesome and you are certain that your friends and co-workers would love to review all of them in detail. You'd like to be courteous and respect that some people will only want to see your cat photos, while others will only want to see your dog photos, but separating the photos is laborious. Let's build a semi-supervised learning system that can classify images of cats and dogs.

Recall from [Chapter 3, Feature Extraction and Preprocessing](#), that a naïve approach to classifying images is to use the intensities, or brightnesses, of all of the pixels as explanatory variables. This approach produces high-dimensional feature vectors for even small

images. Unlike the high-dimensional feature vectors we used to represent documents, these vectors are not sparse. Furthermore, it is obvious that this approach is sensitive to the image's illumination, scale, and orientation. In [Chapter 3](#), *Feature Extraction and Preprocessing*, we also discussed SIFT and SURF descriptors, which describe interesting regions of an image in ways that are invariant to scale, rotation, and illumination. In this example, we will cluster the descriptors extracted from all of the images to learn features. We will then represent an image with a vector with one element for each cluster. Each element will encode the number of descriptors extracted from the image that were assigned to the cluster. This approach is sometimes called the **bag-of-features** representation, as the collection of clusters is analogous to the bag-of-words representation's vocabulary. We will use 1,000 images of cats and 1,000 images of dogs from the training set for Kaggle's *Dogs vs. Cats* competition. The dataset can be downloaded from

<https://www.kaggle.com/c/dogs-vs-cats/data>. We will label cats as the positive class and dogs as the negative class. Note that the images have different dimensions; since our feature vectors do not represent pixels, we do not need to resize the images to have the same dimensions. We will train using the first 60 percent of the images, and test on the remaining 40 percent:

```
>>> import numpy as np
>>> import mahotas as mh
>>> from mahotas.features import surf
>>> from sklearn.linear_model import
LogisticRegression
>>> from sklearn.metrics import *
>>> from sklearn.cluster import MiniBatchKMeans
>>> import glob
```

First, we load the images, convert them to grayscale, and extract the SURF descriptors. SURF descriptors can be extracted more quickly than many similar features, but extracting descriptors from 2,000 images is still computationally expensive. Unlike the previous examples, this script requires several minutes to execute on most computers:

```
>>> all_instance_filenames = []
>>> all_instance_targets = []
>>> for f in glob.glob('cats-and-dogs-
img/*.jpg'):
>>>     target = 1 if 'cat' in f else 0
>>>     all_instance_filenames.append(f)
>>>     all_instance_targets.append(target)
>>> surf_features = []
>>> counter = 0
>>> for f in all_instance_filenames:
>>>     print 'Reading image:', f
>>>     image = mh.imread(f, as_grey=True)
>>>     surf_features.append(surf.surf(image)[:,5:])
```

```
>>> train_len = int(len(all_instance_filenames)
* .60)
>>> X_train_surf_features =
np.concatenate(surf_features[:train_len])
>>> X_test_surf_feautres =
np.concatenate(surf_features[train_len:])
>>> y_train = all_instance_targets[:train_len]
>>> y_test = all_instance_targets[train_len:]
```

We then group the extracted descriptors into 300 clusters in the following code sample. We use `MiniBatchKMeans`, a variation of K-Means that uses a random sample of the instances in each iteration. As it computes the distances to the centroids for only a sample of the instances in each iteration, `MiniBatchKMeans` converges more quickly but its clusters' distortions may be greater. In practice, the results are similar, and this compromise is acceptable.:

```
>>> n_clusters = 300
>>> print 'Clustering',
len(X_train_surf_features), 'features'
>>> estimator =
MiniBatchKMeans(n_clusters=n_clusters)
>>>
estimator.fit_transform(X_train_surf_features)
```

Next, we construct feature vectors for the training and testing data. We find the cluster associated with each of the extracted SURF descriptors, and count them using NumPy's `binCount()` function. The following code

produces a 300-dimensional feature vector for each instance:

```
>>> X_train = []
>>> for instance in surf_features[:train_len]:
>>>     clusters = estimator.predict(instance)
>>>     features = np.bincount(clusters)
>>>     if len(features) < n_clusters:
>>>         features = np.append(features,
np.zeros((1, n_clusters-len(features))))))
>>>     X_train.append(features)

>>> X_test = []
>>> for instance in surf_features[train_len:]:
>>>     clusters = estimator.predict(instance)
>>>     features = np.bincount(clusters)
>>>     if len(features) < n_clusters:
>>>         features = np.append(features,
np.zeros((1, n_clusters-len(features))))))
>>>     X_test.append(features)
```

Finally, we train a logistic regression classifier on the feature vectors and targets, and assess its precision, recall, and accuracy:

```
>>> clf = LogisticRegression(C=0.001,
penalty='l2')
>>> clf.fit_transform(X_train, y_train)
>>> predictions = clf.predict(X_test)
>>> print classification_report(y_test,
predictions)
>>> print 'Precision: ', precision_score(y_test,
predictions)
```

```
>>> print 'Recall: ', recall_score(y_test,  
predictions)  
>>> print 'Accuracy: ', accuracy_score(y_test,  
predictions)
```

Reading image: dog.9344.jpg

...

Reading image: dog.8892.jpg

Clustering 756914 features

		precision	recall	f1-score
	support			
392	0	0.71	0.76	0.73
408	1	0.75	0.70	0.72
800	avg / total	0.73	0.73	0.73

Precision: 0.751322751323

Recall: 0.696078431373

Accuracy: 0.7275

This semi-supervised system has better precision and recall than a logistic regression classifier that uses only the pixel intensities as features. Furthermore, our feature representations have only 300 dimensions; even small 100 x 100 pixel images would have 10,000 dimensions.

Summary

In this chapter, we discussed our first unsupervised learning task: clustering. Clustering is used to discover structure in unlabeled data. You learned about the K-Means clustering algorithm, which iteratively assigns instances to clusters and refines the positions of the cluster centroids. While K-Means learns from experience without supervision, its performance is still measurable; you learned to use distortion and the silhouette coefficient to evaluate clusters. We applied K-Means to two different problems. First, we used K-Means for image quantization, a compression technique that represents a range of colors with a single color. We also used K-Means to learn features in a semi-supervised image classification problem.

In the next chapter, we will discuss another unsupervised learning task called dimensionality reduction. Like the semi-supervised feature representations we created to classify images of cats and dogs, dimensionality reduction can be used to reduce the dimensions of a set of explanatory variables while retaining as much information as possible.

Chapter 7. Dimensionality Reduction with PCA

In this chapter, we will discuss a technique for reducing the dimensions of data called **Principal Component Analysis (PCA)**. Dimensionality reduction is motivated by several problems. First, it can be used to mitigate problems caused by the curse of dimensionality. Second, dimensionality reduction can be used to compress data while minimizing the amount of information that is lost. Third, understanding the structure of data with hundreds of dimensions can be difficult; data with only two or three dimensions can be visualized easily. We will use PCA to visualize a high-dimensional dataset in two dimensions, and build a face recognition system.

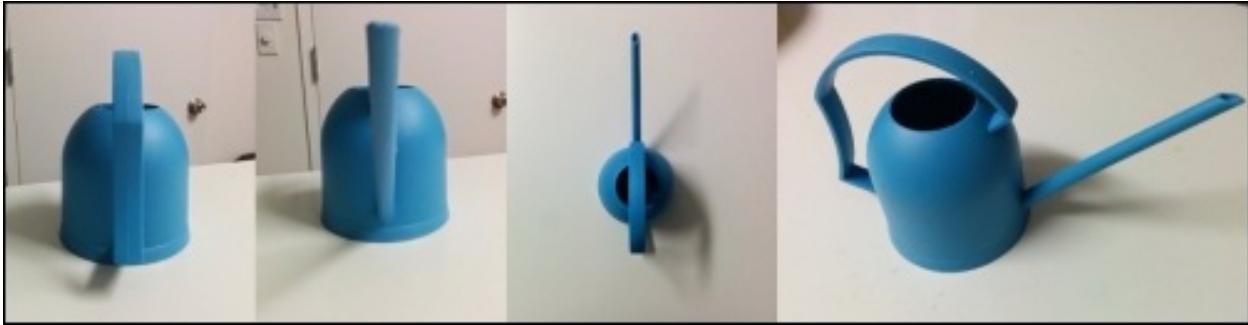
An overview of PCA

Recall from [Chapter 3, Feature Extraction and Preprocessing](#), that problems involving high-dimensional data can be affected by the curse of dimensionality. As the dimensions of a data set increases, the number of samples required for an estimator to generalize increases exponentially. Acquiring such large data may be infeasible in some applications, and learning from large data sets requires more memory and processing power. Furthermore, the sparseness of data often increases with its dimensions. It can become more difficult to detect similar instances in high-dimensional space as all of the instances are similarly sparse.

Principal Component Analysis, also known as the Karhunen-Loeve Transform, is a technique used to search for patterns in high-dimensional data. PCA is commonly used to explore and visualize high-dimensional data sets. It can also be used to compress data, and process data before it is used by another estimator. PCA reduces a set of possibly-correlated, high-dimensional variables to a lower-dimensional set of linearly uncorrelated synthetic variables called **principal components**. The lower-dimensional data will preserve as much of the variance of the original data as possible.

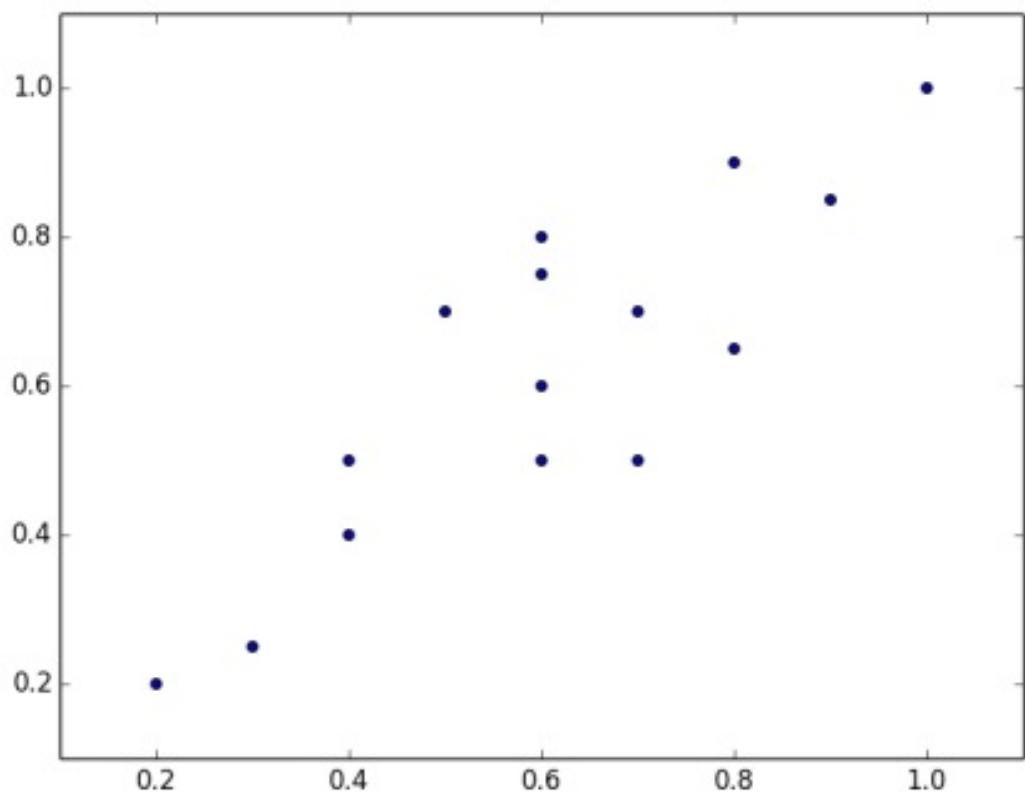
PCA reduces the dimensions of a data set by projecting the data onto a lower-dimensional subspace. For example, a two dimensional data set could be reduced by projecting the points onto a line; each instance in the data set would then be represented by a single value rather than a pair of values. A three-dimensional dataset could be reduced to two dimensions by projecting the variables onto a plane. In general, an n -dimensional dataset can be reduced by projecting the dataset onto a k -dimensional subspace, where k is less than n . More formally, PCA can be used to find a set of vectors that span a subspace, which minimizes the sum of the squared errors of the projected data. This projection will retain the greatest proportion of the original data set's variance.

Imagine that you are a photographer for a gardening supply catalog, and that you are tasked with photographing a watering can. The watering can is three-dimensional, but the photograph is two-dimensional; you must create a two-dimensional representation that describes as much of the watering can as possible. The following are four possible pictures that you could use:

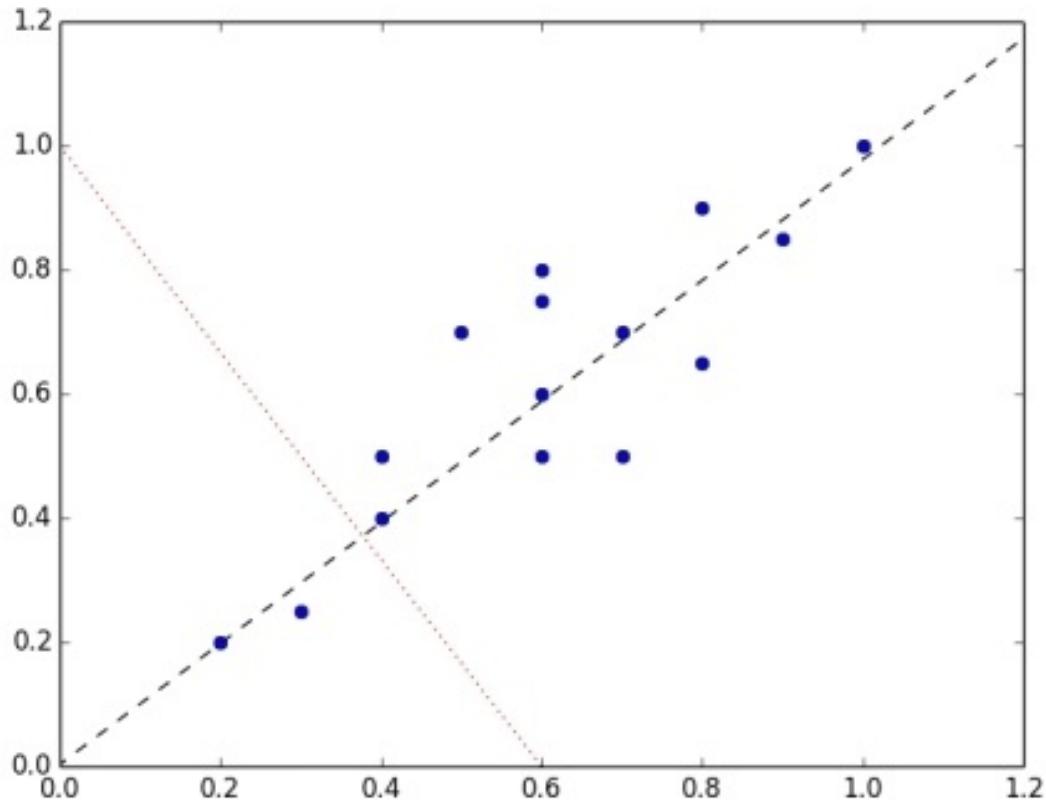


In the first photograph, the back of the watering can is visible, but the front cannot be seen. The second picture is angled to look directly down the spout of the watering can; this picture provides information about the front of the can that was not visible in the first photograph, but now the handle cannot be seen. The height of the watering can cannot be discerned from the bird's eye view of the third picture. The fourth picture is the obvious choice for the catalog; the watering can's height, top, spout, and handle are all discernible in this image.

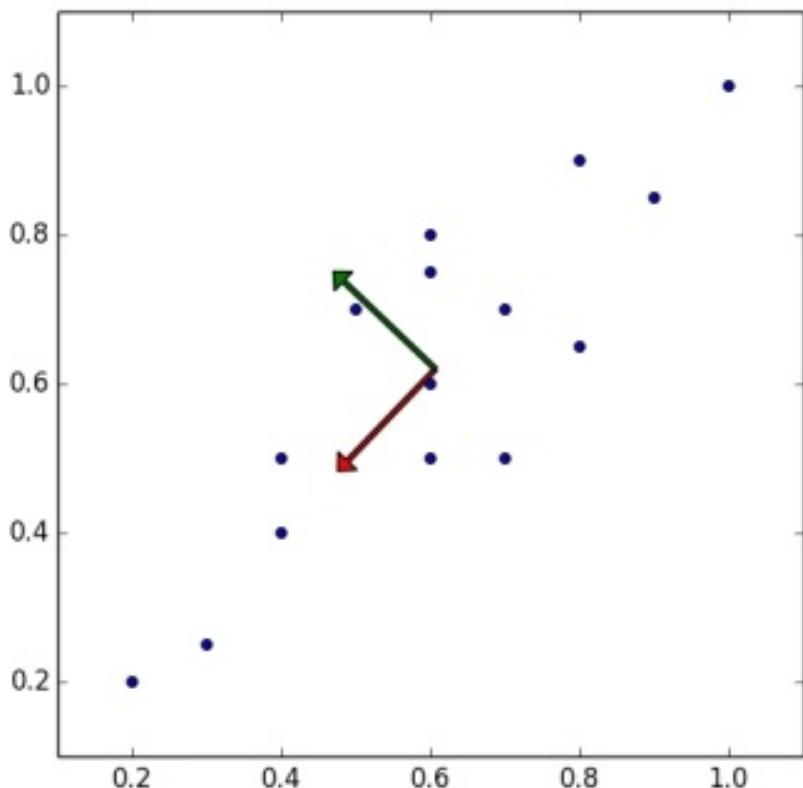
The motivation of PCA is similar; it can project data in a high-dimensional space to a lower-dimensional space that retains as much of the variance as possible. PCA rotates the data set to align with its principal components to maximize the variance contained within the first several principal components. Assume that we have the data set that is plotted in the following figure:



The instances approximately form a long, thin ellipse stretching from the origin to the top right of the plot. To reduce the dimensions of this data set, we must project the points onto a line. The following are two lines that the data could be projected onto. Along which line do the instances vary the most?

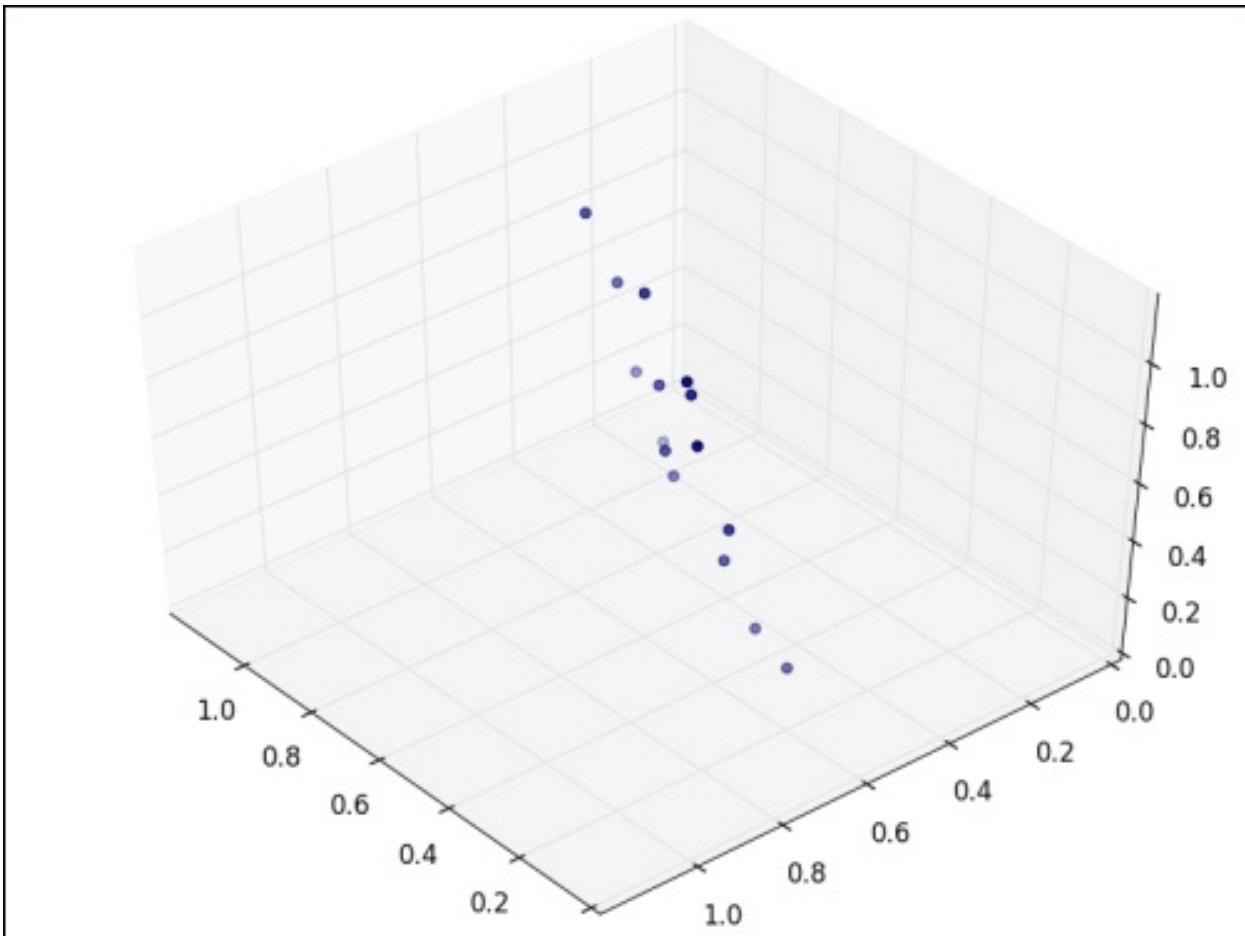


The instances vary more along the dashed line than the dotted line. In fact, the dashed line is the first principal component. The second principal component must be orthogonal to the first principal component; that is, the second principal component must be statistically independent, and will appear to be perpendicular to the first principal component when it is plotted, as shown in the following figure:



Each subsequent principal component preserves the maximum amount of the remaining variance; the only constraint is that each must be orthogonal to the other principal components.

Now assume that the data set is three dimensional. The scatter plot of the points looks like a flat disc that has been rotated slightly about one of the axes.



The points can be rotated and translated such that the tilted disk lies almost exactly in two dimensions. The points now form an ellipse; the third dimension contains almost no variance and can be discarded.

PCA is most useful when the variance in a data set is distributed unevenly across the dimensions. Consider a three-dimensional data set with a spherical convex hull. PCA cannot be used effectively with this data set because

there is equal variance in each dimension; none of the dimensions can be discarded without losing a significant amount of information.

It is easy to visually identify the principal components of data sets with only two or three dimensions. In the next section, we will discuss how to calculate the principal components of high-dimensional data.

Performing Principal Component Analysis

There are several terms that we must define before discussing how principal component analysis works.

Variance, Covariance, and Covariance Matrices

Recall that **variance** is a measure of how a set of values are spread out. Variance is calculated as the average of the squared differences of the values and mean of the values, as per the following equation:

$$s^2 = \frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n-1}$$

Covariance is a measure of how much two variables change together; it is a measure of the strength of the correlation between two sets of variables. If the covariance of two variables is zero, the variables are uncorrelated. Note that uncorrelated variables are not necessarily independent, as correlation is only a measure of linear dependence. The covariance of two variables is calculated using the following equation:

$$\text{cov}(X, Y) = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{y})}{n-1}$$

If the covariance is nonzero, the sign indicates whether the variables are positively or negatively correlated. When two variables are positively correlated, one increases as the other increases. When variables are negatively correlated, one variable decreases relative to its mean as the other variable increases relative to its mean. A **covariance matrix** describes the covariance values between each pair of dimensions in a data set. The element (i, j) indicates the covariance of the i th and j th dimensions of the data. For example, a covariance matrix for a three-dimensional data is given by the following matrix:

$$C = \begin{bmatrix} \text{cov}(x_1, x_1) & \text{cov}(x_1, x_2) & \text{cov}(x_1, x_3) \\ \text{cov}(x_2, x_1) & \text{cov}(x_2, x_2) & \text{cov}(x_2, x_3) \\ \text{cov}(x_3, x_1) & \text{cov}(x_3, x_2) & \text{cov}(x_3, x_3) \end{bmatrix}$$

Let's calculate the covariance matrix for the following data set:

2	0	-1.4
2.2	0.2	-1.5

2.4	0.1	-1
1.9	0	-1.2

The means of the variables are 2.125, 0.075, and -1.275. We can then calculate the covariances of each pair of variables to produce the following covariance matrix:

$$C = \begin{bmatrix} 0.0491666667 & 0.0141666667 & 0.0191666667 \\ 0.0141666667 & 0.00916667 & -0.00583333 \\ 0.0191666667 & -0.00583333 & 0.04916667 \end{bmatrix}$$

We can verify our calculations using NumPy:

```
>>> import numpy as np
>>> X = [[2, 0, -1.4],
>>>       [2.2, 0.2, -1.5],
>>>       [2.4, 0.1, -1],
>>>       [1.9, 0, -1.2]]
>>> print np.cov(np.array(X).T)
[[ 0.04916667  0.01416667  0.01916667]
 [ 0.01416667  0.00916667 -0.00583333]
 [ 0.01916667 -0.00583333  0.04916667]]
```

Eigenvectors and eigenvalues

A vector is described by a **direction** and **magnitude**, or length. An **eigenvector** of a matrix is a non-zero vector that satisfies the following equation:

$$A\vec{v} = \lambda\vec{v}$$

In the preceding equation, \vec{v} is an eigenvector, A is a square matrix, and λ is a scalar called an **eigenvalue**. The direction of an eigenvector remains the same after it has been transformed by A ; only its magnitude has changed, as indicated by the eigenvalue; that is, multiplying a matrix by one of its eigenvectors is equal to scaling the eigenvector. The prefix *eigen* is the German word for *belonging to* or *peculiar to*; the eigenvectors of a matrix are the vectors that *belong* to and characterize the structure of the data.

Eigenvectors and eigenvalues can only be derived from square matrices, and not all square matrices have eigenvectors or eigenvalues. If a matrix does have eigenvectors and eigenvalues, it will have a pair for each of its dimensions. The principal components of a matrix are the eigenvectors of its covariance matrix, ordered by

their corresponding eigenvalues. The eigenvector with the greatest eigenvalue is the first principal component; the second principal component is the eigenvector with the second greatest eigenvalue, and so on.

Let's calculate the eigenvectors and eigenvalues of the following matrix:

$$A = \begin{bmatrix} 1 & -2 \\ 2 & -3 \end{bmatrix}$$

Recall that the product of A and any eigenvector of A must be equal to the eigenvector multiplied by its eigenvalue. We will begin by finding the eigenvalues, which we can find using the following characteristic equations:

$$(A - \lambda I) \vec{v} = 0$$

$$|A - \lambda * I| = \left| \begin{bmatrix} 1 & -2 \\ 2 & -3 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \right| = 0$$

The characteristic equation states that the determinant of the matrix, that is, the difference between the data matrix and the product of the identity matrix and an eigenvalue is zero:

$$\begin{vmatrix} 1-\lambda & -2 \\ 2 & -3-\lambda \end{vmatrix} = (\lambda+1)(\lambda+1) = 0$$

Both of the eigenvalues for this matrix are equal to **-1**. We can now use the eigenvalues to solve the eigenvectors:

$$A\vec{v} = \lambda\vec{v}$$

First, we set the equation equal to zero:

$$(A - \lambda I)\vec{v} = 0$$

Substituting our values for A produces the following:

$$\left(\begin{bmatrix} 1 & -2 \\ 2 & -3 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \right) \vec{v} = \begin{bmatrix} 1-\lambda & -2 \\ 2 & -3-\lambda \end{bmatrix} \vec{v} = \begin{bmatrix} 1-\lambda & -2 \\ 2 & -3-\lambda \end{bmatrix} \begin{bmatrix} v_{1,1} \\ v_{1,2} \end{bmatrix} = 0$$

We can then substitute the first eigenvalue in our first eigenvalue to solve the eigenvectors.

$$\begin{bmatrix} 1 - (-1) & -2 \\ 2 & -3 - (-1) \end{bmatrix} \begin{bmatrix} v_{1,1} \\ v_{1,2} \end{bmatrix} = \begin{bmatrix} 2 & -2 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} v_{1,1} \\ v_{1,2} \end{bmatrix} = 0$$

The preceding equation can be rewritten as a system of equations:

$$\begin{cases} 2v_{1,1} + -(2v_{1,2}) = 0 \\ 2v_{1,1} + -(2v_{1,2}) = 0 \end{cases}$$

Any non-zero vector that satisfies the preceding equations, such as the following, can be used as an eigenvector:

$$\begin{bmatrix} 1 & -2 \\ 2 & -3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = -1 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

PCA requires unit eigenvectors, or eigenvectors that have a length equal to 1. We can normalize an eigenvector by dividing it by its norm, which is given by the following equation:

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$$

The norm of our vector is equal to the following:

$$\left\| \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\| = \sqrt{1^2 + 1^2} = \sqrt{2}$$

This produces the following unit eigenvector:

$$\frac{\begin{bmatrix} 1 \\ 1 \end{bmatrix}}{\sqrt{2}} = \begin{bmatrix} 0.7071067811865475 \\ 0.7071067811865475 \end{bmatrix}$$

We can verify that our solutions for the eigenvectors are correct using NumPy. The `eig` function returns a tuple of the eigenvalues and eigenvectors:

```
>>> import numpy as np
>>> w, v = np.linalg.eig(np.array([[1, -2], [2,
-3]]))
>>> w; v
array([-0.99999998, -1.0000002])
array([[ 0.70710678,  0.70710678],
```

Dimensionality reduction with Principal Component Analysis

Let's use principal component analysis to reduce the following two-dimensional data set to one dimension:

x1	x2
0.9	1
2.4	2.6
1.2	1.7
0.5	0.7
0.3	0.7
1.8	1.4
0.5	0.6
0.3	0.6
2.5	2.6

1.3	1.1

The first step of PCA is to subtract the mean of each explanatory variable from each observation:

x1	x2
$0.9 - 1.17 = -0.27$	$1 - 1.3 = -0.3$
$2.4 - 1.17 = 1.23$	$2.6 - 1.3 = 1.3$
$1.2 - 1.17 = 0.03$	$1.7 - 1.3 = 0.4$
$0.5 - 1.17 = -0.67$	$-0.7 - 1.3 = 0.6$
$0.3 - 1.17 = -0.87$	$-0.7 - 1.3 = 0.6$
$1.8 - 1.17 = 0.63$	$1.4 - 1.3 = 0.1$
$0.5 - 1.17 = -0.67$	$0.6 - 1.3 = -0.7$
$0.3 - 1.17 = -0.87$	$0.6 - 1.3 = -0.7$
$2.5 - 1.17 = 1.33$	$2.6 - 1.3 = 1.3$

$$\boxed{1.3 - 1.17 = 0.13} \quad \boxed{1.1 - 1.3 = -0.2}$$

Next, we must calculate the principal components of the data. Recall that the principal components are the eigenvectors of the data's covariance matrix ordered by their eigenvalues. The principal components can be found using two different techniques. The first technique requires calculating the covariance matrix of the data. Since the covariance matrix will be square, we can calculate the eigenvectors and eigenvalues using the approach described in the previous section. The second technique uses singular value decomposition of the data matrix to find the eigenvectors and square roots of the eigenvalues of the covariance matrix. We will work through an example using the first technique, and then describe the second technique that is used by scikit-learn's implementation of PCA.

The following matrix is the covariance matrix for the data:

$$C = \begin{bmatrix} 0.6867777778 & 0.6066666667 \\ 0.6066666667 & 0.5977777778 \end{bmatrix}$$

Using the technique described in the previous section, the

eigenvalues are 1.250 and 0.034. The following are the unit eigenvectors:

$$\begin{bmatrix} 0.73251454 & -0.68075138 \\ 0.68075138 & 0.73251454 \end{bmatrix}$$

Next, we will project the data onto the principal components. The first eigenvector has the greatest eigenvalue and is the first principal component. We will build a transformation matrix in which each column of the matrix is the eigenvector for a principal component. If we were reducing a five-dimensional data set to three dimensions, we would build a matrix with three columns. In this example, we will project our two-dimensional data set onto one dimension, so we will use only the eigenvector for the first principal component. Finally, we will find the dot product of the data matrix and transformation matrix. The following is the result of projecting our data onto the first principal component:

$$\begin{bmatrix} -0.27 & -0.3 \\ 1.23 & 1.3 \\ 0.03 & 0.4 \\ -0.67 & -0.6 \\ -0.87 & -0.6 \\ 0.63 & 0.1 \\ -0.67 & -0.7 \\ -0.87 & -0.7 \\ 1.33 & 1.3 \\ 0.13 & -0.2 \end{bmatrix} = \begin{bmatrix} -0.40200434 \\ 1.78596968 \\ 0.29427599 \\ -0.89923557 \\ 0.73251454 \\ 0.68075138 \\ -1.04573848 \\ 0.5295593 \\ -0.96731071 \\ -1.11381362 \\ 1.85922113 \\ -0.04092339 \end{bmatrix}$$

Many implementations of PCA, including the one of scikit-learn, use singular value decomposition to calculate the eigenvectors and eigenvalues. SVD is given by the following equation:

$$X = U \Sigma V^T$$

The columns of U are called left singular vectors of the data matrix, the columns of V are its right singular vectors, and the diagonal entries of Σ are its singular values. While the singular vectors and values of a matrix

are useful in some applications of signal processing and statistics, we are only interested in them as they relate to the eigenvectors and eigenvalues of the data matrix. Specifically, the left singular vectors are the eigenvectors of the covariance matrix and the diagonal elements of Σ are the square roots of the eigenvalues of the covariance matrix. Calculating SVD is beyond the scope of this chapter; however, eigenvectors found using SVD should be similar to those derived from a covariance matrix.

Using PCA to visualize high-dimensional data

It is easy to discover patterns by visualizing data with two or three dimensions. A high-dimensional dataset cannot be represented graphically, but we can still gain some insights into its structure by reducing it to two or three principal components.

Collected in 1936, Fisher's Iris data set is a collection of fifty samples from each of the three species of Iris: Iris setosa, Iris virginica, and Iris versicolor. The explanatory variables are measurements of the length and width of the petals and sepals of the flowers. The Iris dataset is commonly used to test classification models, and is included with scikit-learn. Let's reduce the `iris` dataset's four dimensions so that we can visualize it in two dimensions:

```
>>> import matplotlib.pyplot as plt  
>>> from sklearn.decomposition import PCA  
>>> from sklearn.datasets import load_iris
```

First, we load the built-in `iris` data set and instantiate a `PCA` estimator. The `PCA` class takes a number of principal components to retain as a hyperparameter. Like the other

estimators, PCA exposes a `fit_transform()` method that returns the reduced data matrix:

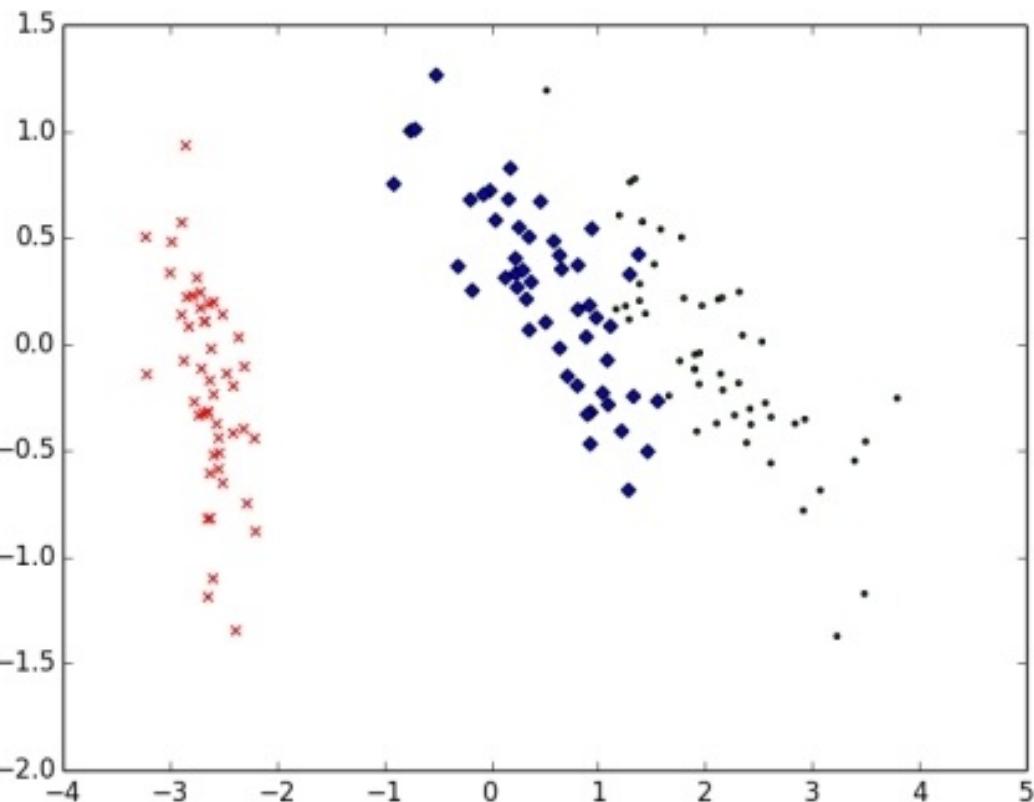
```
>>> data = load_iris()
>>> y = data.target
>>> X = data.data
>>> pca = PCA(n_components=2)
>>> reduced_X = pca.fit_transform(X)
```

Finally, we assemble and plot the reduced data:

```
>>> red_x, red_y = [], []
>>> blue_x, blue_y = [], []
>>> green_x, green_y = [], []
>>> for i in range(len(reduced_X)):
>>>     if y[i] == 0:
>>>         red_x.append(reduced_X[i][0])
>>>         red_y.append(reduced_X[i][1])
>>>     elif y[i] == 1:
>>>         blue_x.append(reduced_X[i][0])
>>>         blue_y.append(reduced_X[i][1])
>>>     else:
>>>         green_x.append(reduced_X[i][0])
>>>         green_y.append(reduced_X[i][1])
>>> plt.scatter(red_x, red_y, c='r', marker='x')
>>> plt.scatter(blue_x, blue_y, c='b',
marker='D')
>>> plt.scatter(green_x, green_y, c='g',
marker='.')
>>> plt.show()
```

The reduced instances are plotted in the following figure.

Each of the dataset's three classes is indicated by its own marker style. From this two-dimensional view of the data, it is clear that one of the classes can be easily separated from the other two overlapping classes. It would be difficult to notice this structure without a graphical representation. This insight can inform our choice of classification model.



Face recognition with PCA

Now let's apply PCA to a face-recognition problem. Face recognition is the supervised classification task of identifying a person from an image of his or her face. In this example, we will use a data set called *Our Database of Faces* from AT&T Laboratories, Cambridge. The data set contains ten images each of forty people. The images were created under different lighting conditions, and the subjects varied their facial expressions. The images are gray scale and 92 x 112 pixels in dimension. The following is an example image:



While these images are small, a feature vector that encodes the intensity of every pixel will have 10,304 dimensions. Training from such high-dimensional data could require many samples to avoid over-fitting. Instead, we will use PCA to compactly represent the images in

terms of a small number of principal components.

We can reshape the matrix of pixel intensities for an image into a vector, and create a matrix of these vectors for all of the training images. Each image is a linear combination of this data set's principal components. In the context of face recognition, these principal components are called **eigenfaces**. The eigenfaces can be thought of as standardized components of faces. Each face in the data set can be expressed as some combination of the eigenfaces, and can be approximated as a combination of the most important eigenfaces:

```
>>> from os import walk, path
>>> import numpy as np
>>> import mahotas as mh
>>> from sklearn.cross_validation import
train_test_split
>>> from sklearn.cross_validation import
cross_val_score
>>> from sklearn.preprocessing import scale
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import
LogisticRegression
>>> from sklearn.metrics import
classification_report
>>> X = []
>>> y = []
```

We begin by loading the images into NumPy arrays, and

reshaping their matrices into vectors:

```
>>> for dir_path, dir_names, file_names in  
walk('data/att-faces/orl_faces'):  
>>>     for fn in file_names:  
>>>         if fn[-3:] == 'pgm':  
>>>             image_filename =  
path.join(dir_path, fn)  
>>>  
X.append(scale(mh.imread(image_filename,  
as_grey=True).reshape(10304).astype('float32')))  
>>>             y.append(dir_path)  
>>> X = np.array(X)
```

We then randomly split the images into training and test sets, and fit the `PCA` object on the training set:

```
>>> X_train, X_test, y_train, y_test =  
train_test_split(X, y)  
>>> pca = PCA(n_components=150)
```

We reduce all of the instances to 150 dimensions and train a logistic regression classifier. The data set contains forty classes; scikit-learn automatically creates binary classifiers using the one versus all strategy behind the scenes:

```
>>> X_train_reduced = pca.fit_transform(X_train)  
>>> X_test_reduced = pca.transform(X_test)  
>>> print 'The original dimensions of the  
training data were', X_train.shape  
>>> print 'The reduced dimensions of the
```

```
training data are', X_train_reduced.shape  
>>> classifier = LogisticRegression()  
>>> accuracies = cross_val_score(classifier,  
X_train_reduced, y_train)
```

Finally, we evaluate the performance of the classifier using cross-validation and a test set. The average per-class F1 score of the classifier trained on the full data was 0.94, but required significantly more time to train and could be prohibitively slow in an application with more training instances:

```
>>> print 'Cross validation accuracy:',  
np.mean(accuracies), accuracies  
>>> classifier.fit(X_train_reduced, y_train)  
>>> predictions =  
classifier.predict(X_test_reduced)  
>>> print classification_report(y_test,  
predictions)
```

The following is the output of the script:

```
The original dimensions of the training data  
were (300, 10304)  
The reduced dimensions of the training data are  
(300, 150)  
Cross validation accuracy: 0.833841819347 [  
0.82882883 0.83 0.84269663]  
precision recall f1-score  
support  
data/att-faces/orl_faces/s1 1.00 1.00
```

1.00	2		
data/att-faces/orl_faces/s10		1.00	
1.00	1.00	2	
data/att-faces/orl_faces/s11			1.00
0.60	0.75	5	
...			
data/att-faces/orl_faces/s9		1.00	1.00
1.00	2		
avg / total	0.92	0.89	0.89
100			

Summary

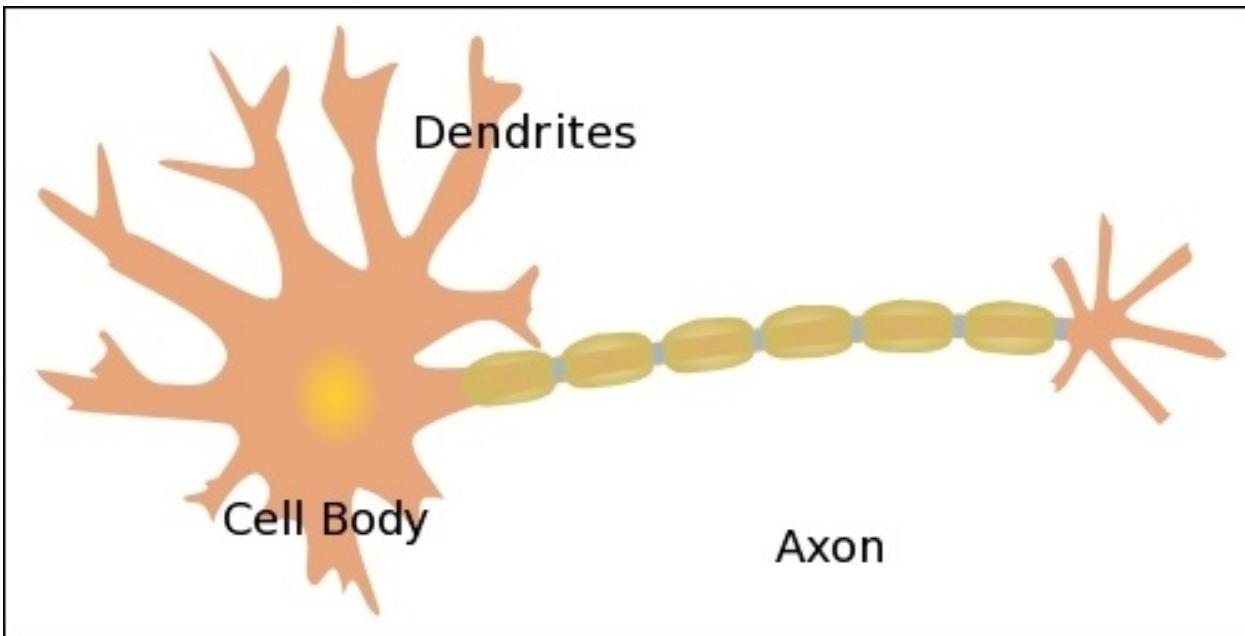
In this chapter, we examined the problem of dimensionality reduction. High-dimensional data cannot be visualized easily. High-dimensional data sets may also suffer from the curse of dimensionality; estimators require many samples to learn to generalize from high-dimensional data. We mitigated these problems using a technique called principal component analysis, which reduces a high-dimensional, possibly-correlated data set to a lower-dimensional set of uncorrelated principal components by projecting the data onto a lower-dimensional subspace. We used principal component analysis to visualize the four-dimensional Iris data set in two dimensions, and build a face-recognition system. In the next chapter, we will return to supervised learning. We will discuss an early classification algorithm called the perceptron, which will prepare us to discuss more advanced models in the last few chapters.

Chapter 8. The Perceptron

In previous chapters we discussed generalized linear models that relate a linear combination of explanatory variables and model parameters to a response variable using a link function. In this chapter, we will discuss another linear model called the perceptron. The perceptron is a binary classifier that can learn from individual training instances, which can be useful for training from large datasets. More importantly, the perceptron and its limitations inspire the models that we will discuss in the final chapters.

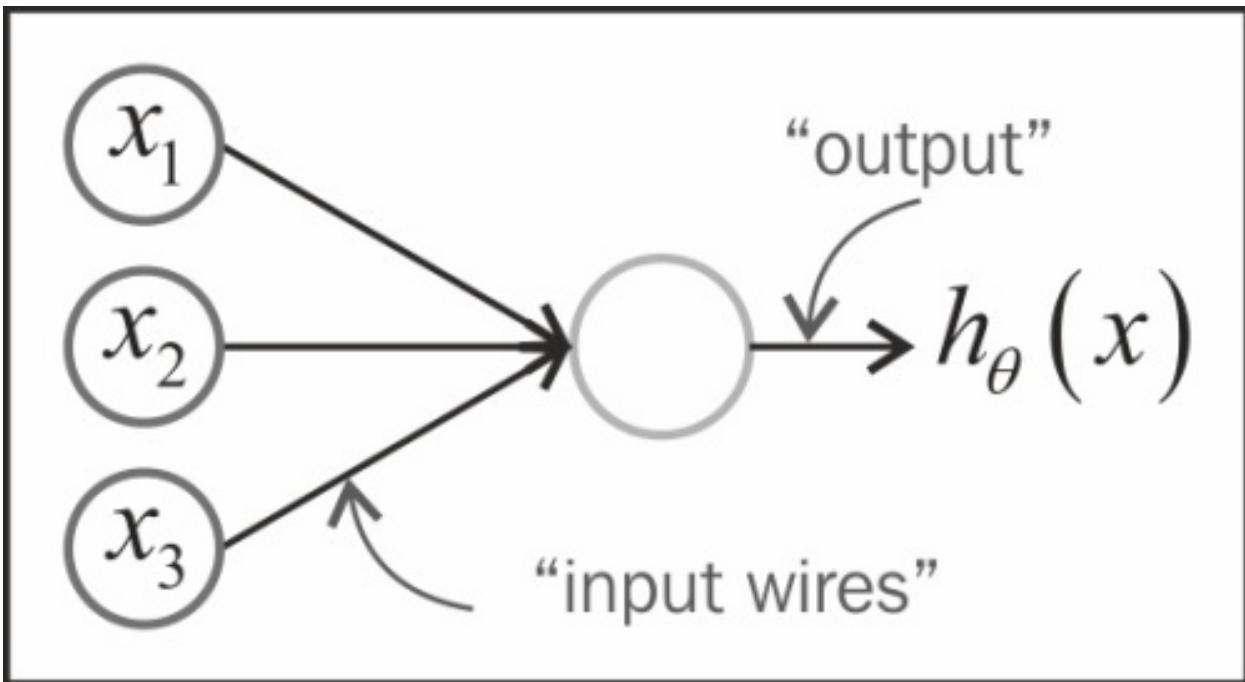
Invented by Frank Rosenblatt at the Cornell Aeronautical Laboratory in the late 1950's, the development of the perceptron was originally motivated by efforts to simulate the human brain. A brain is composed of cells called **neurons** that process information and connections between neurons called **synapses** through which information is transmitted. It is estimated that human brain is composed of as many as 100 billion neurons and 100 trillion synapses. As shown in the following image, the main components of a neuron are dendrites, a body, and an axon. The dendrites receive electrical signals from other neurons. The signals are processed in the neuron's body, which then sends a signal through the axon to

another neuron.



An individual neuron can be thought of as a computational unit that processes one or more inputs to produce an output. A perceptron functions analogously to a neuron; it accepts one or more inputs, processes them, and returns an output. It may seem that a model of just one of the hundreds of billions of neurons in the human brain will be of limited use. To an extent that is true; the perceptron cannot approximate some basic functions. However, we will still discuss perceptrons for two reasons. First, perceptrons are capable of online, error-driven learning; the learning algorithm can update the model's parameters using a single training instance rather

than the entire batch of training instances. Online learning is useful for learning from training sets that are too large to be represented in memory. Second, understanding how the perceptron works is necessary to understand some of the more powerful models that we will discuss in subsequent chapters, including support vector machines and artificial neural networks. Perceptrons are commonly visualized using a diagram like the following one:



The circles labeled x_1 , x_2 , and x_3 are input units. Each input unit represents one feature. Perceptrons frequently use an additional input unit that represents a constant bias term, but this input unit is usually omitted from diagrams.

The circle in the center is a computational unit or the neuron's body. The edges connecting the input units to the computational unit are analogous to dendrites. Each edge is **weighted**, or associated with a parameter. The parameters can be interpreted easily; an explanatory variable that is correlated with the positive class will have a positive weight, and an explanatory variable that is correlated with the negative class will have a negative weight. The edge directed away from the computational unit returns the output and can be thought of as the axon.

Activation functions

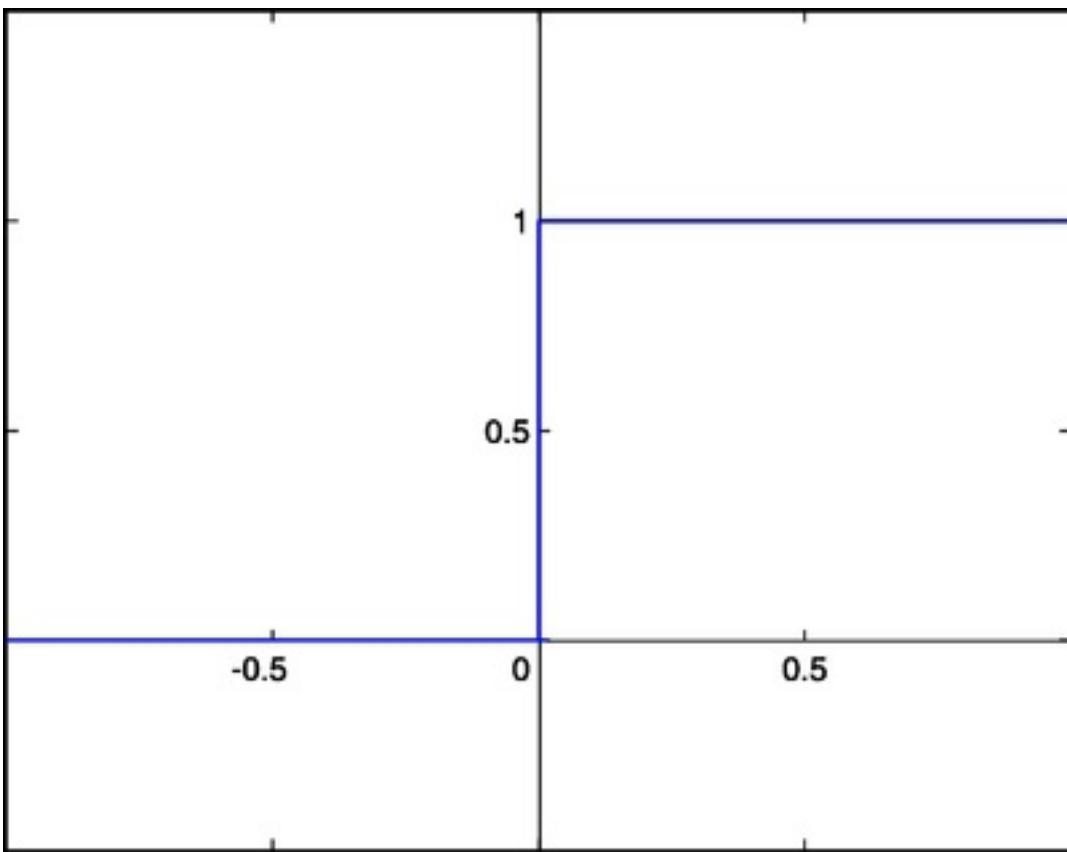
The perceptron classifies instances by processing a linear combination of the explanatory variables and the model parameters using an **activation function** as shown in the following equation. The linear combination of the parameters and inputs is sometimes called the perceptron's **preactivation**.

$$y = \phi \left(\sum_{i=1}^n w_i x_i + b \right)$$

Here, w_i are the model's parameters, b is a constant bias term, and ϕ is the activation function. Several different activation functions are commonly used. Rosenblatt's original perceptron used the **Heaviside step** function. Also called the unit step function, the Heaviside step function is shown in the following equation, where x is the weighted combination of the features:

$$g(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{elsewhere} \end{cases}$$

If the weighted sum of the explanatory variables and the bias term is greater than zero, the activation function returns one and the perceptron predicts that the instance is the positive class. Otherwise, the function returns zero and the perceptron predicts that the instance is the negative class. The Heaviside step activation function is plotted in the following figure:



Another common activation function is the **logistic sigmoid** activation function. The gradients for this activation function can be calculated efficiently, which

will be important in later chapters when we construct artificial neural networks. The logistic sigmoid activation function is given by the following equation, where x is the sum of the weighted inputs:

$$g(x) = \frac{1}{1+e^{-x}}$$

This model should seem familiar; it is a linear combination of the values of the explanatory variables and the model parameters processed through the logistic function. That is, this is identical to the model for logistic regression. While a perceptron with a logistic sigmoid activation function has the same model as logistic regression, it learns its parameters differently.

The perceptron learning algorithm

The perceptron learning algorithm begins by setting the weights to zero or to small random values. It then predicts the class for a training instance. The perceptron is an **error-driven** learning algorithm; if the prediction is correct, the algorithm continues to the next instance. If the prediction is incorrect, the algorithm updates the weights. More formally, the update rule is given by the following:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i}, \text{ for all feature } 0 \leq i \leq n$$

For each training instance, the value of the parameter for each explanatory variable is incremented by $\alpha(d_j - y_j(t))x_{j,i}$, where d_j is the true class for instance j , $y_j(t)$ is the predicted class for instance j , $x_{j,i}$ is the value of the i^{th} explanatory variable for instance j , and α is a hyperparameter that controls the learning rate. If the prediction is correct, $d_j - y_j(t)$ equals zero, and the $\alpha(d_j - y_j(t))x_{j,i}$ term equals zero. So, if the prediction is correct, the weight is not updated. If the prediction is

incorrect, the weight is incremented by the product of the learning rate, $d_j - y_j(t)$, and the value of the feature.

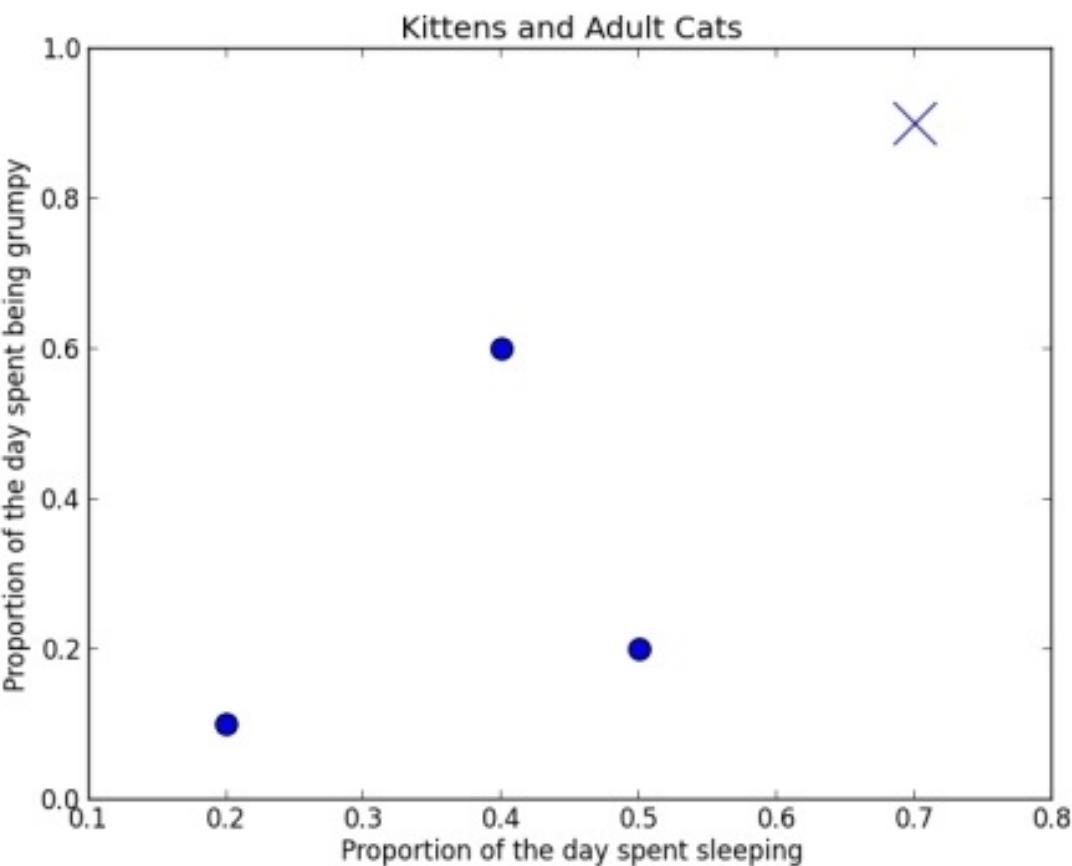
This update rule is similar to the update rule for gradient descent in that the weights are adjusted towards classifying the instance correctly and the size of the update is controlled by a learning rate. Each pass through the training instances is called an **epoch**. The learning algorithm has converged when it completes an epoch without misclassifying any of the instances. The learning algorithm is not guaranteed to converge; later in this chapter, we will discuss linearly inseparable datasets for which convergence is impossible. For this reason, the learning algorithm also requires a hyperparameter that specifies the maximum number of epochs that can be completed before the algorithm terminates.

Binary classification with the perceptron

Let's work through a toy classification problem. Suppose that you wish to separate adult cats from kittens. Only two explanatory variables are available in your dataset: the proportion of the day that the animal was asleep and the proportion of the day that the animal was grumpy. Our training data consists of the following four instances:

Instance	Proportion of the day spent sleeping	Proportion of the day spent being grumpy	Kitten or Adult?
1	0.2	0.1	Kitten
2	0.4	0.6	Kitten
3	0.5	0.2	Kitten
4	0.7	0.9	Adult

The following scatter plot of the instances confirms that they are linearly separable:



Our goal is to train a perceptron that can classify animals using the two real-valued explanatory variables. We will represent kittens with the positive class and adult cats with the negative class. The preceding network diagram describes the perceptron that we will train.

Our perceptron has three input units. x_1 is the input unit for the bias term. x_2 and x_3 are input units for the two features. Our perceptron's computational unit uses a

Heaviside activation function. In this example, we will set the maximum number of training epochs to ten; if the algorithm does not converge within 10 epochs, it will stop and return the current values of the weights. For simplicity, we will set the learning rate to one. Initially, we will set all of the weights to zero. Let's examine the first training epoch, which is shown in the following table:

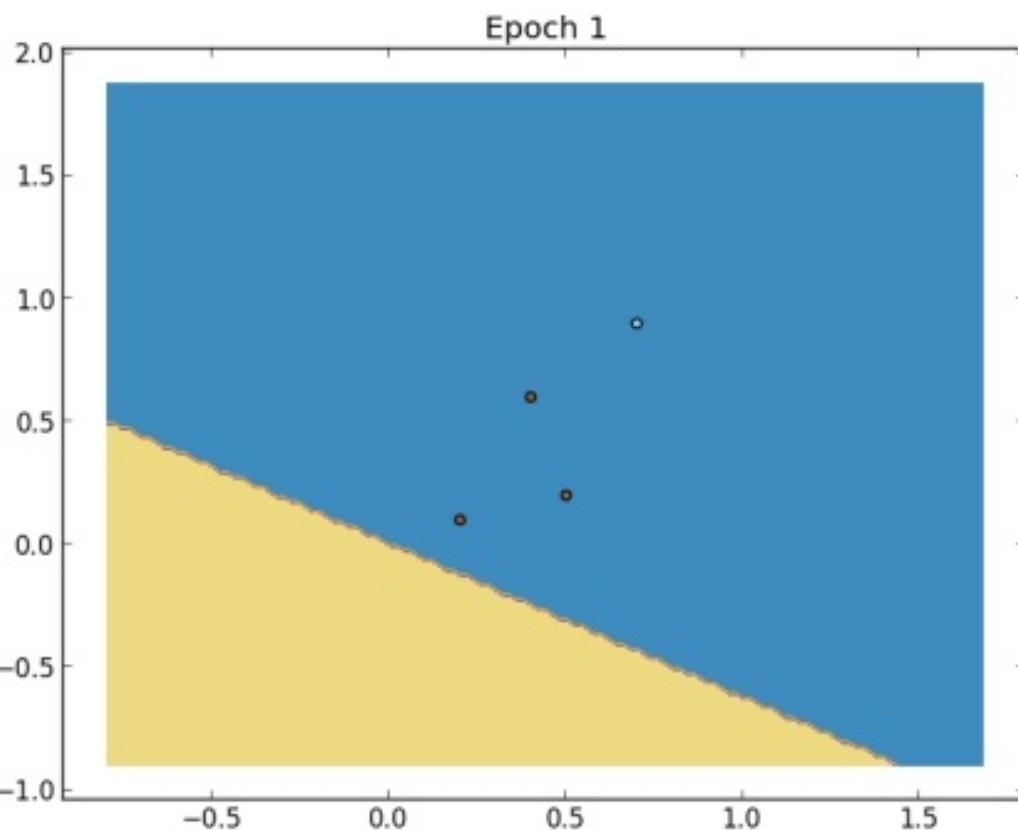
Epoch 1				
Instance	Initial Weights x	Prediction, Target	Correct	Updated weights
	Activation			
0	0, 0, 0; 1.0, 0.2, 0.1; $1.0*0 + 0.2*0 + 0.1*0 = 0.0;$	0, 1	False	1.0, 0.2, 0.1
1	1.0, 0.2, 0.1; 1.0, 0.4, 0.6; $1.0*1.0 + 0.4*0.2 + 0.6*0.1 = 1.14;$	1, 1	True	1.0, 0.2, 0.1

	1.0, 0.2, 0.1; 1.0, 0.5, 0.2; 1.0*1.0 + 0.5*0.2 + 0.2*0.1 = 1.12;	1, 1	True	1.0, 0.2, 0.1
3	1.0, 0.2, 0.1; 1.0, 0.7, 0.9; 1.0*1.0 + 0.7*0.2 + 0.9*0.1 = 1.23;	1, 0	False	0, -0.5, -0.8

Initially, all of the weights are equal to zero. The weighted sum of the explanatory variables for the first instance is zero, the activation function outputs zero, and the perceptron incorrectly predicts that the kitten is an adult cat. As the prediction was incorrect, we update the weights according to the update rule. We increment each of the weights by the product of the learning rate, the difference between the true and predicted labels and the value of the corresponding feature.

We then continue to the second training instance and calculate the weighted sum of its features using the updated weights. This sum equals 1.14, so the activation function outputs one. This prediction is correct, so we continue to the third training instance without updating

the weights. The prediction for the third instance is also correct, so we continue to the fourth training instance. The weighted sum of the features for the fourth instance is 1.23. The activation function outputs one, incorrectly predicting that this adult cat is a kitten. Since this prediction is incorrect, we increment each weight by the product of the learning rate, the difference between the true and predicted labels, and its corresponding feature. We completed the first epoch by classifying all of the instances in the training set. The perceptron did not converge; it classified half of the training instances incorrectly. The following figure depicts the decision boundary after the first epoch:



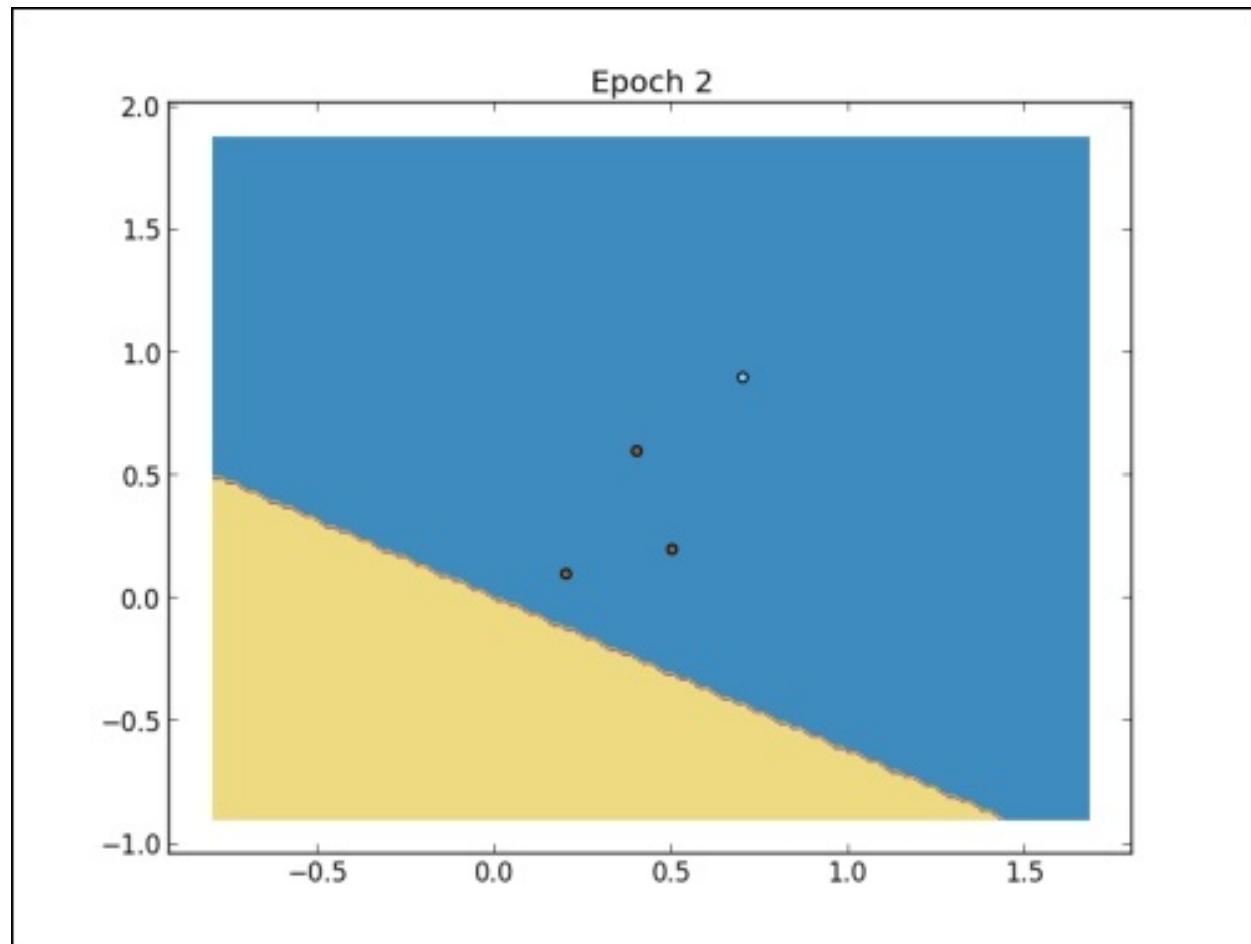
Note that the decision boundary moved throughout the epoch; the decision boundary formed by the weights at the end of the epoch would not necessarily have produced the same predictions seen earlier in the epoch. Since we have not exceeded the maximum number of training epochs, we will iterate through the instances again. The second training epoch is shown in the following table:

--	--	--	--	--

Epoch 2				
Instance	Initial Weights x	Prediction, Target	Correct	Updated weights
	Activation			
0	0, -0.5, -0.8 1.0, 0.2, 0.1 $1.0*0 + 0.2*(-0.5) + 0.1*(-0.8) = -0.18$	0, 1	False	1, -0.3, -0.7
1	1, -0.3, -0.7 1.0, 0.4, 0.6 $1.0*1.0 + 0.4*(-0.3) + 0.6*(-0.7) = 0.46$	1, 1	True	1, -0.3, -0.7
2	1, -0.3, -0.7 1.0, 0.5, 0.2 $1.0*1.0 + 0.5*(-0.3) + 0.2*(-0.7) = 0.71$	1, 1	True	1, -0.3, -0.7
3	1, -0.3, -0.7 1.0, 0.7, 0.9 $1.0*1.0 + 0.7*(-0.3) + 0.9*(-0.7) = -0.8$	1, 0	False	0, -1, -1.6

$$0.9 * -0.7 = 0.16$$

The second epoch begins using the values of the weights from the first epoch. Two training instances are classified incorrectly during this epoch. The weights are updated twice, but the decision boundary at the end of the second epoch is similar the decision boundary at the end of the first epoch.

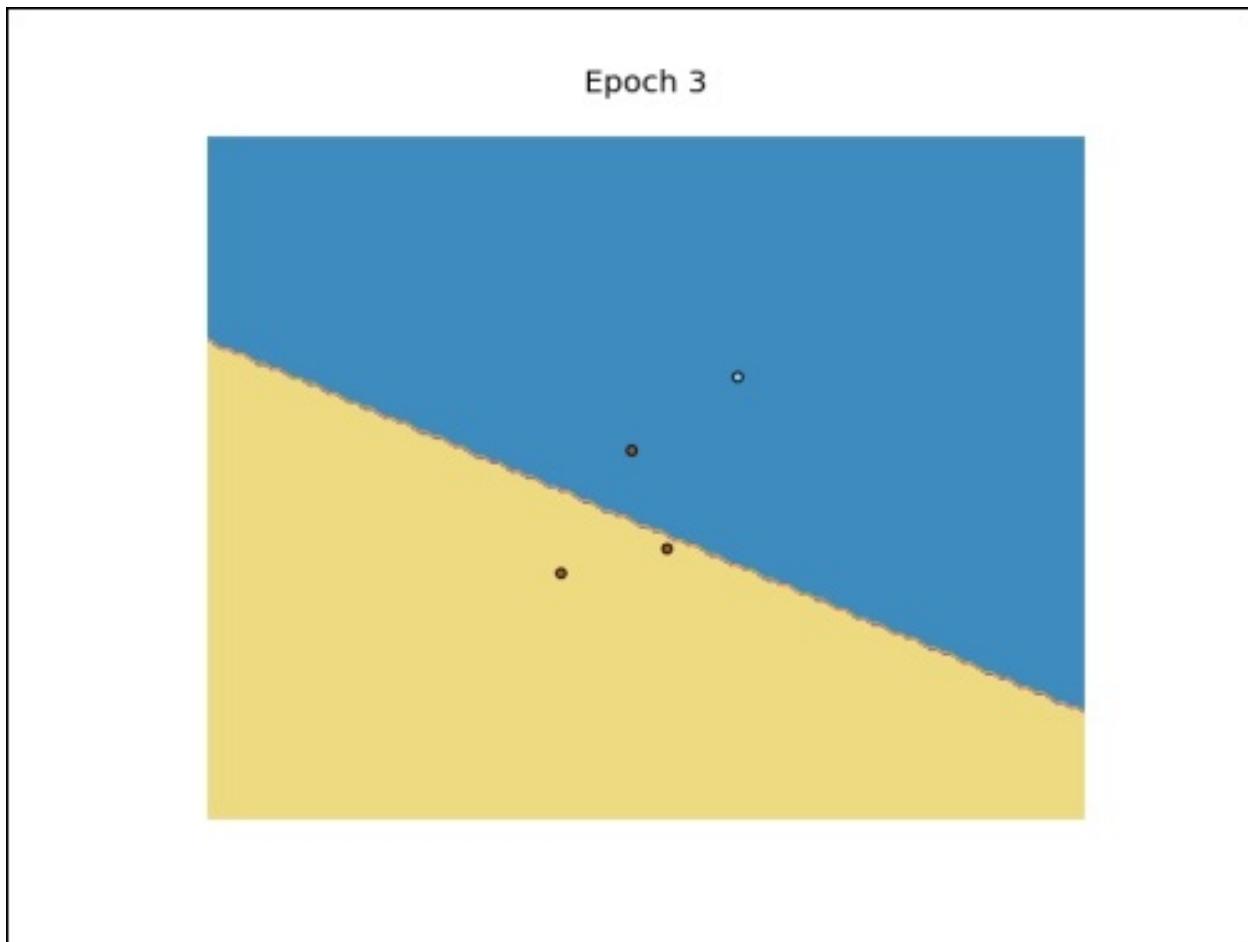


The algorithm failed to converge during this epoch, so we will continue training. The following table describes the third training epoch:

Epoch 3				
Instance	Initial Weights x	Prediction, Target	Correct	Updated Weights
	Activation			
0	0, -1, -1.6 1.0, 0.2, 0.1 $1.0*0 + 0.2*(-1.0) + 0.1*(-1.6) = -0.36$	0, 1	False	1, -0.8, -1.5
1	1, -0.8, -1.5 1.0, 0.4, 0.6 $1.0*1.0 + 0.4*(-0.8) + 0.6*(-1.5) = -0.22$	0, 1	False	2, -0.4, -0.9
2	2, -0.4, -0.9 1.0, 0.5, 0.2 $1.0*2.0 + 0.5*(-0.4) + 0.2*(-0.9) = 1.62$	1, 1	True	2, -0.4, -0.9

	2, -0.4, -0.9			
3	1.0, 0.7, 0.9	1, 0	False	1, -1.1, -1.8
	1.0*2.0 + 0.7*-0.4 + 0.9*-0.9 = 0.91			

The perceptron classified more instances incorrectly during this epoch than during previous epochs. The following figure depicts the decision boundary at the end of the third epoch:



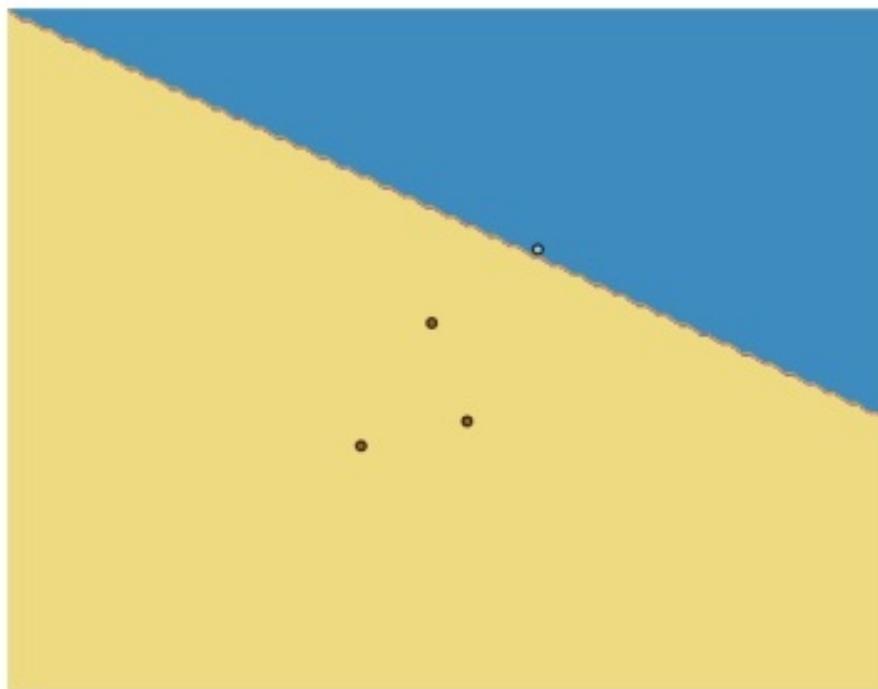
The perceptron continues to update its weights throughout the fourth and fifth training epochs, and it continues to classify training instances incorrectly. During the sixth epoch the perceptron classified all of the instances correctly; it converged on a set of weights that separates the two classes. The following table describes the sixth training epoch:

Epoch 6					
Instance	x	Initial Weights	Prediction, Target	Correct	
		Activation		Updated weights	
0		$\begin{aligned} &2, -1, -1.5 \\ &1.0, 0.2, 0.1 \\ &1.0*2 + 0.2*-1 + 0.1*-1.5 \\ &= 1.65 \end{aligned}$	1, 1	True	2, -1, -1.5
1		$\begin{aligned} &2, -1, -1.5 \\ &1.0, 0.4, 0.6 \\ &1.0*2 + 0.4*-1 + 0.6*-1.5 \\ &= 0.70 \end{aligned}$	1, 1	True	2, -1, -1.5

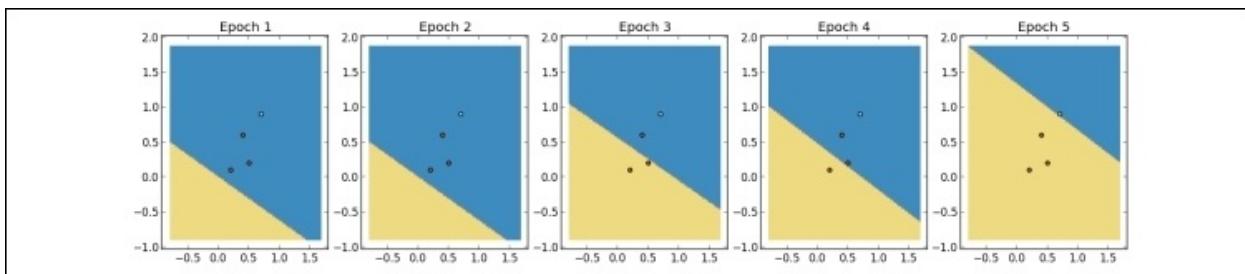
	$2, -1, -1.5$ $1.0, 0.5, 0.2$ $1.0*2 + 0.5*-1 + 0.2*-1.5$ $= 1.2$	$1, 1$	True	$2, -1, -1.5$
3	$2, -1, -1.5$ $1.0, 0.7, 0.9$ $1.0*2 + 0.7*-1 + 0.9*-1.5$ $= -0.05$	$0, 0$	True	$2, -1, -1.5$

The decision boundary at the end of the sixth training epoch is shown in the following figure:

Epoch 6



The following figure shows the decision boundary throughout all the training epochs.



Document classification with the perceptron

scikit-learn provides an implementation of the perceptron. As with the other implementations that we used, the constructor for the `Perceptron` class accepts keyword arguments that set the algorithm's hyperparameters.

`Perceptron` similarly exposes the `fit_transform()` and `predict()` methods. `Perceptron` also provides a `partial_fit()` method, which allows the classifier to train and make predictions for streaming data.

In this example, we train a perceptron to classify documents from the 20 newsgroups dataset. The dataset consists of approximately 20,000 documents sampled from 20 Usenet newsgroups. The dataset is commonly used in document classification and clustering experiments; scikit-learn provides a convenience function to download and read the dataset. We will train a perceptron to classify documents from three newsgroups: `rec.sports.hockey`, `rec.sports.baseball`, and `rec.auto`. scikit-learn's `Perceptron` natively supports multiclass classification; it will use the one versus all strategy to train a classifier for each of the classes in the training data. We will represent the documents as TF-

IDF-weighted bags of words. The `partial_fit()` method could be used in conjunction with `HashingVectorizer` to train from large or streaming data in a memory-constrained setting:

```
>>> from sklearn.datasets import
fetch_20newsgroups
>>> from sklearn.metrics import
f1_score, classification_report
>>> from sklearn.feature_extraction.text import
TfidfVectorizer
>>> from sklearn.linear_model import Perceptron

>>> categories = ['rec.sport.hockey',
'rec.sport.baseball', 'rec.autos']
>>> newsgroups_train =
fetch_20newsgroups(subset='train',
categories=categories, remove=('headers',
'footers', 'quotes'))
>>> newsgroups_test =
fetch_20newsgroups(subset='test',
categories=categories, remove=('headers',
'footers', 'quotes'))

>>> vectorizer = TfidfVectorizer()
>>> X_train =
vectorizer.fit_transform(newsgroups_train.data)
>>> X_test =
vectorizer.transform(newsgroups_test.data)

>>> classifier = Perceptron(n_iter=100,
eta0=0.1)
>>> classifier.fit(X_train,
```

```

newsgroups_train.target )
>>> predictions = classifier.predict(X_test)
>>> print
classification_report(newsgroups_test.target,
predictions)

```

The following is the output of the script:

		precision	recall	f1-score
	support			
396	0	0.89	0.87	0.88
397	1	0.87	0.78	0.82
399	2	0.79	0.88	0.83
1192	avg / total	0.85	0.85	0.85

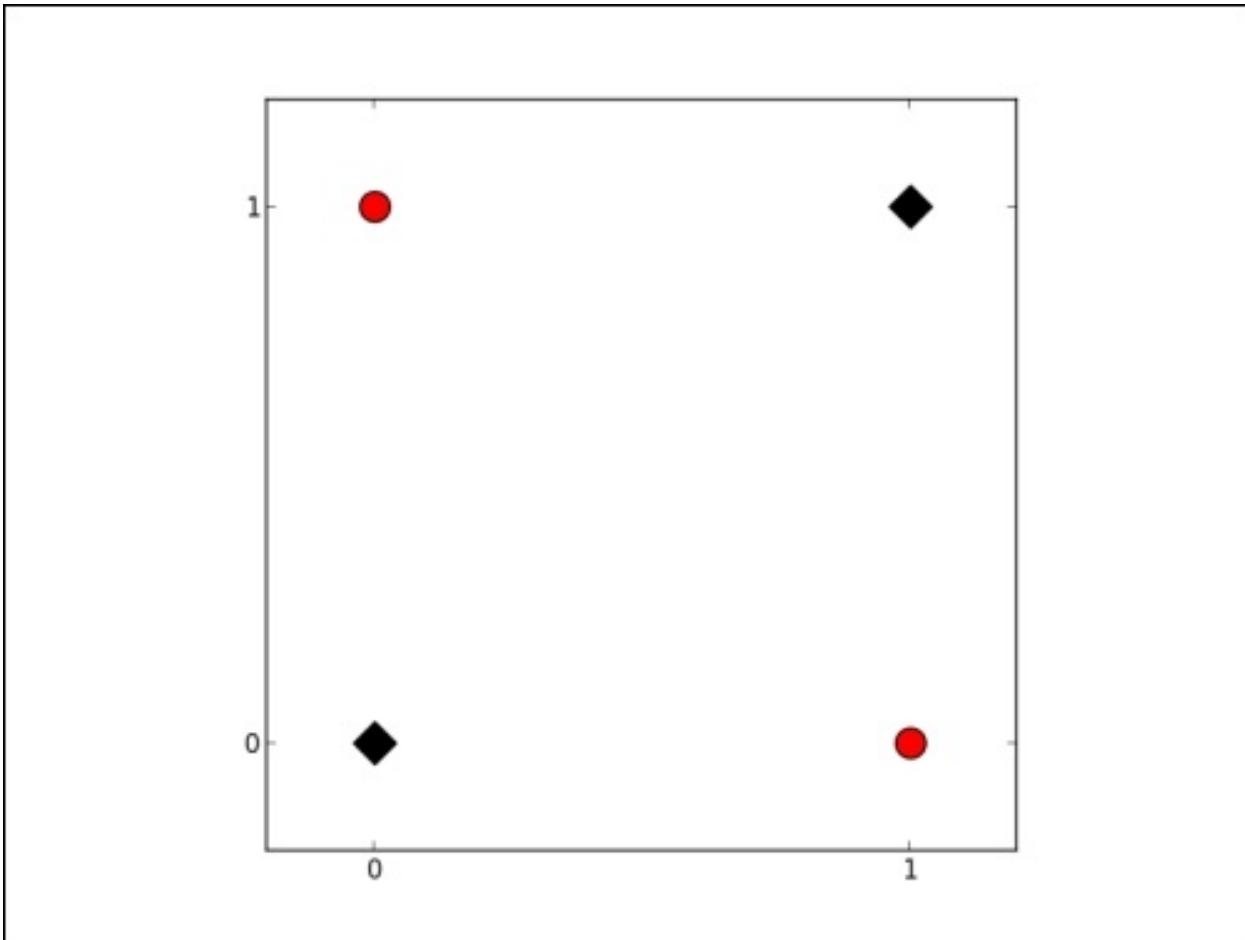
First, we download and read the dataset using the `fetch_20newsgroups()` function. Consistent with other built-in datasets, the function returns an object with `data`, `target`, and `target_names` fields. We also specify that the documents' headers, footers, and quotes should be removed. Each of the newsgroups used different conventions in the headers and footers; retaining these explanatory variables makes classifying the documents artificially easy. We produce TF-IDF vectors using `TfidfVectorizer`, train the perceptron, and evaluate it on

the test set. Without hyperparameter optimization, the perceptron's average precision, recall, and F1 score are 0.85.

Limitations of the perceptron

While the perceptron classified the instances in our example well, the model has limitations. Linear models like the perceptron with a Heaviside activation function are not **universal function approximators**; they cannot represent some functions. Specifically, linear models can only learn to approximate the functions for **linearly separable** datasets. The linear classifiers that we have examined find a hyperplane that separates the positive classes from the negative classes; if no hyperplane exists that can separate the classes, the problem is not linearly separable.

A simple example of a function that is linearly inseparable is the logical operation **XOR**, or exclusive disjunction. The output of XOR is one when one of its inputs is equal to one and the other is equal to zero. The inputs and outputs of XOR are plotted in two dimensions in the following graph. When XOR outputs **1**, the instance is marked with a circle; when XOR outputs **0**, the instance is marked with a diamond, as shown in the following figure:



It is impossible to separate the circles from the diamonds using a single straight line. Suppose that the instances are pegs on a board. If you were to stretch a rubber band around both of the positive instances, and stretch a second rubber band around both of the negative instances, the bands would intersect in the middle of the board. The rubber bands represent **convex hulls**, or the envelope that contains all of the points within the set and all of the

points along any line connecting a pair points within the set. Feature representations are more likely to be linearly separable in higher dimensional spaces than lower dimensional spaces. For instance, text classification problems tend to be linearly separable when high-dimensional representations like the bag-of-words are used.

In the next two chapters, we will discuss techniques that can be used to model linearly inseparable data. The first technique, called **kernelization**, projects linearly inseparable data to a higher dimensional space in which it is linearly separable. Kernelization can be used in many models, including perceptrons, but it is particularly associated with support vector machines, which we will discuss in the next chapter. Support vector machines also support techniques that can find the hyperplane that separates linearly inseparable classes with the fewest errors. The second technique creates a directed graph of perceptrons. The resulting model, called an **artificial neural network**, is a universal function approximator; we will discuss artificial neural networks in [Chapter 10, From the Perceptron to Artificial Neural Networks](#).

Summary

In this chapter, we discussed the perceptron. Inspired by neurons, the perceptron is a linear model for binary classification. The perceptron classifies instances by processing a linear combination of the explanatory variables and weights with an activation function. While a perceptron with a logistic sigmoid activation function is the same model as logistic regression, the perceptron learns its weights using an online, error-driven algorithm. The perceptron can be used effectively in some problems. Like the other linear classifiers that we have discussed, the perceptron is not a universal function approximator; it can only separate the instances of one class from the instances of the other using a hyperplane. Some datasets are not linearly separable; that is, no possible hyperplane can classify all of the instances correctly. In the following chapters, we will discuss two models that can be used with linearly inseparable data: the artificial neural network, which creates a universal function approximator from a graph of perceptrons and the support vector machine, which projects the data onto a higher dimensional space in which it is linearly separable.

Chapter 9. From the Perceptron to Support Vector Machines

In the previous chapter we discussed the perceptron. As a binary classifier, the perceptron cannot be used to effectively classify linearly inseparable feature representations. We encountered a similar problem to this in our discussion of multiple linear regression in [Chapter 2, Linear Regression](#); we examined a dataset in which the response variable was not linearly related to the explanatory variables. To improve the accuracy of the model, we introduced a special case of multiple linear regression called polynomial regression. We created synthetic combinations of features, and were able to model a linear relationship between the response variable and the features in the higher-dimensional feature space.

While this method of increasing the dimensions of the feature space may seem like a promising technique to use when approximating nonlinear functions with linear models, it suffers from two related problems. The first is a computational problem; computing the mapped features and working with larger vectors requires more computing

power. The second problem pertains to generalization; increasing the dimensions of the feature representation introduces the curse of dimensionality. Learning from high-dimensional feature representations requires exponentially more training data to avoid overfitting.

In this chapter, we will discuss a powerful model for classification and regression called the **support vector machine (SVM)**. First, we will revisit mapping features to higher-dimensional spaces. Then, we will discuss how support vector machines mitigate the computation and generalization problems encountered when learning from the data mapped to higher-dimensional spaces. Entire books are devoted to describing support vector machines, and describing the optimization algorithms used to train SVMs requires more advanced math than we have used in previous chapters. Instead of working through toy examples in detail as we have done in previous chapters, we will try to develop an intuition for how support vector machines work in order to apply them effectively with scikit-learn.

Kernels and the kernel trick

Recall that the perceptron separates the instances of the positive class from the instances of the negative class using a hyperplane as a decision boundary. The decision boundary is given by the following equation:

$$f(x) = \langle w, x \rangle + b$$

Predictions are made using the following function:

$$h(x) = \text{sign}(f(x))$$

Note that previously we expressed the inner product $\langle w, x \rangle$ as $w^T x$. To be consistent with the notational conventions used for support vector machines, we will adopt the former notation in this chapter.

While the proof is beyond the scope of this chapter, we can write the model differently. The following expression

of the model is called the **dual** form. The expression we used previously is the **primal** form:

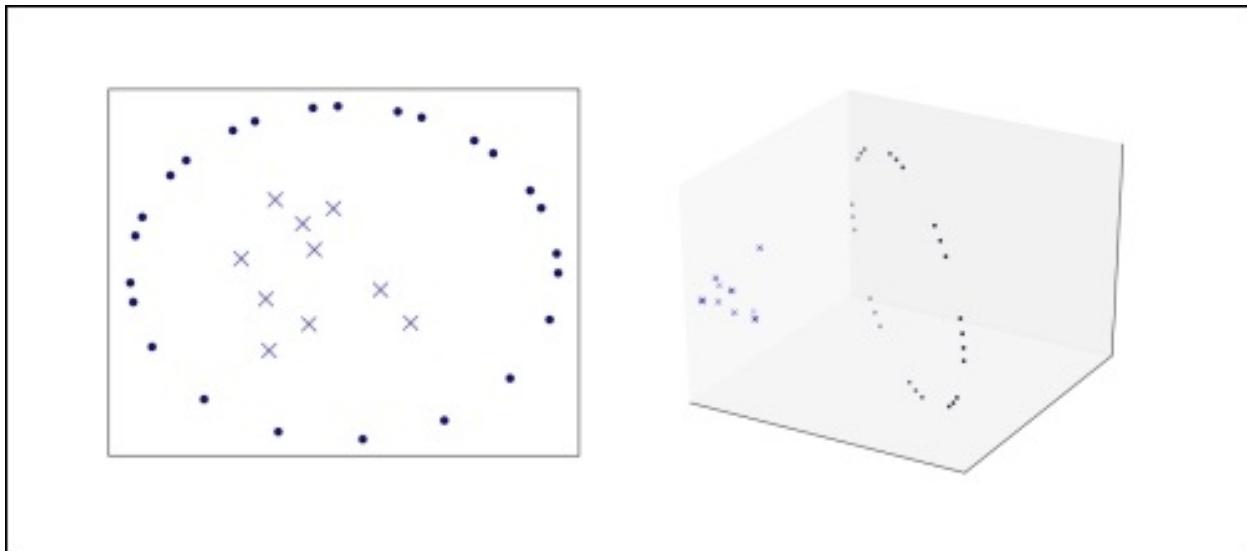
$$f(x) = \langle w, x \rangle + b = \sum \alpha_i y_i \langle x_i, x \rangle + b$$

The most important difference between the primal and dual forms is that the primal form computes the inner product of the *model parameters* and the test instance's feature vector, while the dual form computes the inner product of the *training instances* and the test instance's feature vector. Shortly, we will exploit this property of the dual form to work with linearly inseparable classes. First, we must formalize our definition of mapping features to higher-dimensional spaces.

In the section on polynomial regression in [Chapter 2](#), *Linear Regression*, we mapped features to a higher-dimensional space in which they were linearly related to the response variable. The mapping increased the number of features by creating quadratic terms from combinations of the original features. These synthetic features allowed us to express a nonlinear function with a linear model. In general, a mapping is given by the following expression:

$$x \rightarrow \phi(x)$$
$$\phi: R^d \rightarrow R^D$$

The plot on the left in the following figure shows the original feature space of a linearly inseparable data set. The plot on the right shows that the data is linearly separable after mapping to a higher-dimensional space:



Let's return to the dual form of our decision boundary, and the observation that the feature vectors appear only inside of a dot product. We could map the data to a higher-dimensional space by applying the mapping to the feature vectors as follows:

$$f(x) = \sum \alpha_i y_i \langle x_i, x \rangle + b$$

$$f(x) = \sum \alpha_i y_i \langle \phi(x_i) \phi(x) \rangle + b$$

As noted, this mapping allows us to express more complex models, but it introduces computation and generalization problems. Mapping the feature vectors and computing their dot products can require a prohibitively large amount of processing power.

Observe in the second equation that while we have mapped the feature vectors to a higher-dimensional space, the feature vectors still only appear as a dot product. The dot product is scalar; we do not require the mapped feature vectors once this scalar has been computed. If we can use a different method to produce the same scalar as the dot product of the mapped vectors, we can avoid the costly work of explicitly computing the dot product and mapping the feature vectors.

Fortunately, there is such a method called the **kernel trick**. A **kernel** is a function that, given the original

feature vectors, returns the same value as the dot product of its corresponding mapped feature vectors. Kernels do not explicitly map the feature vectors to a higher-dimensional space, or calculate the dot product of the mapped vectors. Kernels produce the same value through a different series of operations that can often be computed more efficiently. Kernels are defined more formally in the following equation:

$$K(x, z) = \langle \phi(x), \phi(z) \rangle$$

Let's demonstrate how kernels work. Suppose that we have two feature vectors, x and z :

$$x = (x_1, x_2)$$

$$z = (z_1, z_2)$$

In our model, we wish to map the feature vectors to a higher-dimensional space using the following transformation:

$$\phi(x) = x^2$$

The dot product of the mapped, normalized feature vectors is equivalent to:

$$\langle \phi(x), \phi(z) \rangle = \left\langle \left(x_1^2, x_2^2, \sqrt{2}x_1x_2 \right), \left(z_1^2, z_2^2, \sqrt{2}z_1z_2 \right) \right\rangle$$

The kernel given by the following equation produces the same value as the dot product of the mapped feature vectors:

$$K(x, z) = \langle x, z \rangle^2 = (x_1z_1 + x_2z_2)^2 = x_1^2z_1^2 + 2x_1z_1x_2z_2 + x_2^2z_2^2$$

$$K(x, z) = \langle \phi(x), \phi(z) \rangle$$

Let's plug in values for the feature vectors to make this example more concrete:

$$x = (4, 9)$$

$$z = (3, 3)$$

$$K(x, z) = 4^2 * 3^2 + 2 * 4 * 3 * 9 * 3 = 1521$$

$$\langle \phi(x), \phi(z) \rangle = \left\langle (4^2, 9^2, \sqrt{2} * 4 * 9), (3^2, 3^2, \sqrt{2} * 3 * 3) \right\rangle = 1521$$

The kernel $K(x, z)$ produced the same value as the dot product $\langle \phi(x), \phi(z) \rangle$ of the mapped feature vectors, but never explicitly mapped the feature vectors to the higher-dimensional space and required fewer arithmetic operations. This example used only two dimensional feature vectors. Data sets with even a modest number of features can result in mapped feature spaces with massive dimensions. scikit-learn provides several commonly used kernels, including the polynomial, sigmoid, Gaussian, and linear kernels. Polynomial kernels are given by the

following equation:

$$K(x, x') = (1 + x \times x')^k$$

Quadratic kernels, or polynomial kernels where k is equal to 2, are commonly used in natural language processing.

The sigmoid kernel is given by the following equation. γ and r are hyperparameters that can be tuned through cross-validation:

$$K(x, x') = \tanh \langle \gamma(x, x') + r \rangle$$

The Gaussian kernel is a good first choice for problems requiring nonlinear models. The Gaussian kernel is a **radial basis function**. A decision boundary that is a hyperplane in the mapped feature space is similar to a decision boundary that is a hypersphere in the original space. The feature space produced by the Gaussian kernel can have an infinite number of dimensions, a feat that would be impossible otherwise. The Gaussian kernel is given by the following equation:

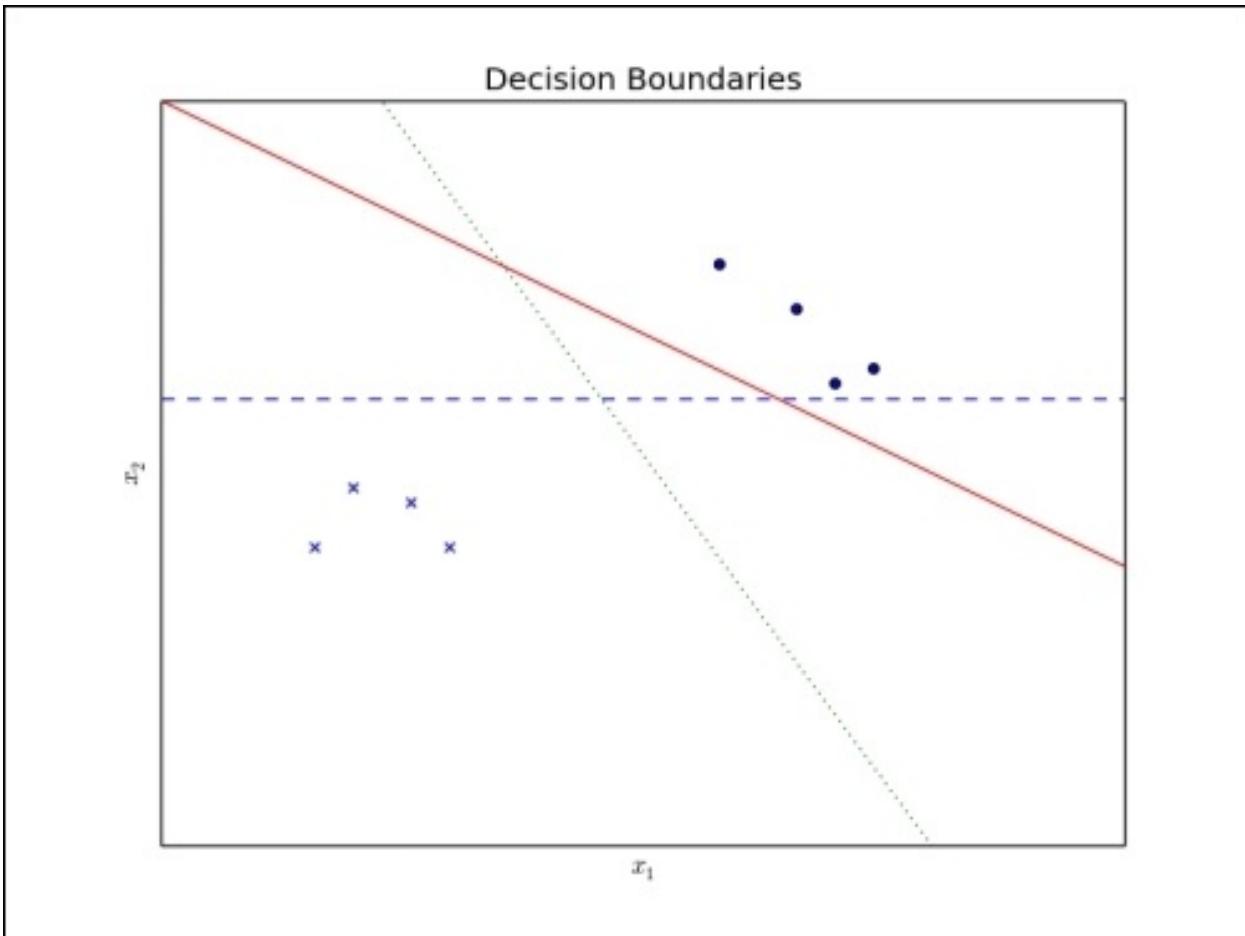
$$K(x, x') = e^{-\|x - x'\|^2 / \sigma^2}$$

γ is a hyperparameter. It is always important to scale the features when using support vector machines, but feature scaling is especially important when using the Gaussian kernel.

Choosing a kernel can be challenging. Ideally, a kernel will measure the similarity between instances in a way that is useful to the task. While kernels are commonly used with support vector machines, they can also be used with any model that can be expressed in terms of the dot product of two feature vectors, including logistic regression, perceptrons, and principal component analysis. In the next section, we will address the second problem caused by mapping to high-dimensional feature spaces: generalization.

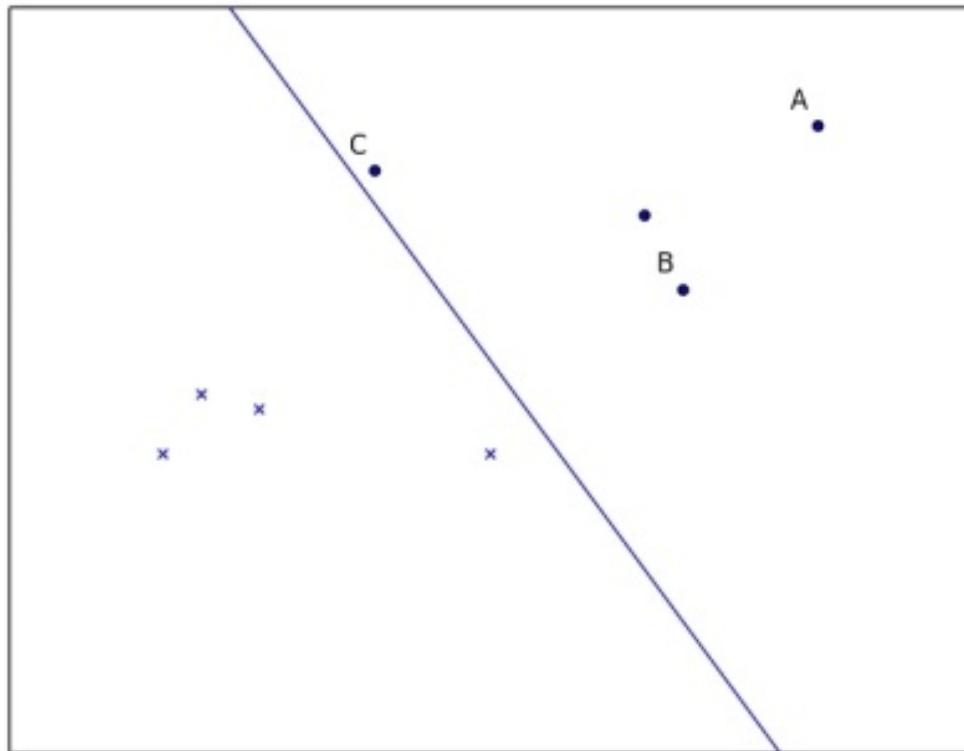
Maximum margin classification and support vectors

The following figure depicts instances from two linearly separable classes and three possible decision boundaries. All of the decision boundaries separate the training instances of the positive class from the training instances of the negative class, and a perceptron could learn any of them. Which of these decision boundaries is most likely to perform best on test data?



From this visualization, it is intuitive that the dotted decision boundary is the best. The solid decision boundary is near many of the positive instances. The test set could contain a positive instance that has a slightly smaller value for the first explanatory variable, x_1 ; this instance would be classified incorrectly. The dashed decision boundary is farther away from most of the training instances; however, it is near one of the positive instances and one of the negative instances. The following

figure provides a different perspective on evaluating decision boundaries:



Assume that the line plotted is the decision boundary for a logistic regression classifier. The instance labeled **A** is far from the decision boundary; it would be predicted to belong to the positive class with a high probability. The instance labeled **B** would still be predicted to belong to the positive class, but the probability would be lower as

the instance is closer to the decision boundary. Finally, the instance labeled **C** would be predicted to belong to the positive class with a low probability; even a small change to the training data could change the class that is predicted. The most confident predictions are for the instances that are farthest from the decision boundary. We can estimate the confidence of the prediction using its **functional margin**. The functional margin of the training set is given by the following equations:

$$funct = \min y_i f(x_i)$$

$$f(x) = \langle w, x \rangle + b$$

In the preceding formulae y_i is the true class of the instance. The functional margin is large for instance **A** and small for instance **C**. If **C** were misclassified, the functional margin would be negative. The instances for which the functional margin is equal to one are called **support vectors**. These instances alone are sufficient to define the decision boundary; the other instances are not required to predict the class of a test instance. Related to the functional margin is the **geometric margin**, or the

maximum width of the band that separates the support vectors. The geometric margin is equal to the normalized functional margin. It is necessary to normalize the functional margins as they can be scaled by using w , which is problematic for training. When w is a unit vector, the geometric margin is equal to the functional vector. We can now formalize our definition of the best decision boundary as having the greatest geometric margin. The model parameters that maximize the geometric margin can be solved through the following constrained optimization problem:

$$\min \frac{1}{n} \langle w, w \rangle$$

$$\text{subject to: } y_i (\langle w, x_i \rangle + b) \geq 1$$

A useful property of support vector machines is that this optimization problem is convex; it has a single local minimum that is also the global minimum. While the proof is beyond the scope of this chapter, the previous optimization problem can be written using the dual form of the model to accommodate kernels as follows:

$$W(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

$$\text{subject to : } \sum_{i=1}^n y_i \alpha_i = 0$$

$$\text{subject to : } \alpha_i \geq 0$$

Finding the parameters that maximize the geometric margin subject to the constraints that all of the positive instances have functional margins of at least 1 and all of the negative instances have functional margins of at most -1 is a quadratic programming problem. This problem is commonly solved using an algorithm called **Sequential Minimal Optimization (SMO)**. The SMO algorithm breaks the optimization problem down into a series of the smallest possible subproblems, which are then solved analytically.

Classifying characters in scikit-learn

Let's apply support vector machines to a classification problem. In recent years, support vector machines have been used successfully in the task of character recognition. Given an image, the classifier must predict the character that is depicted. Character recognition is a component of many optical character-recognition systems. Even small images require high-dimensional representations when raw pixel intensities are used as features. If the classes are linearly inseparable and must be mapped to a higher-dimensional feature space, the dimensions of the feature space can become even larger. Fortunately, SVMs are suited to working with such data efficiently. First, we will use scikit-learn to train a support vector machine to recognize handwritten digits. Then, we will work on a more challenging problem: recognizing alphanumeric characters in photographs.

Classifying handwritten digits

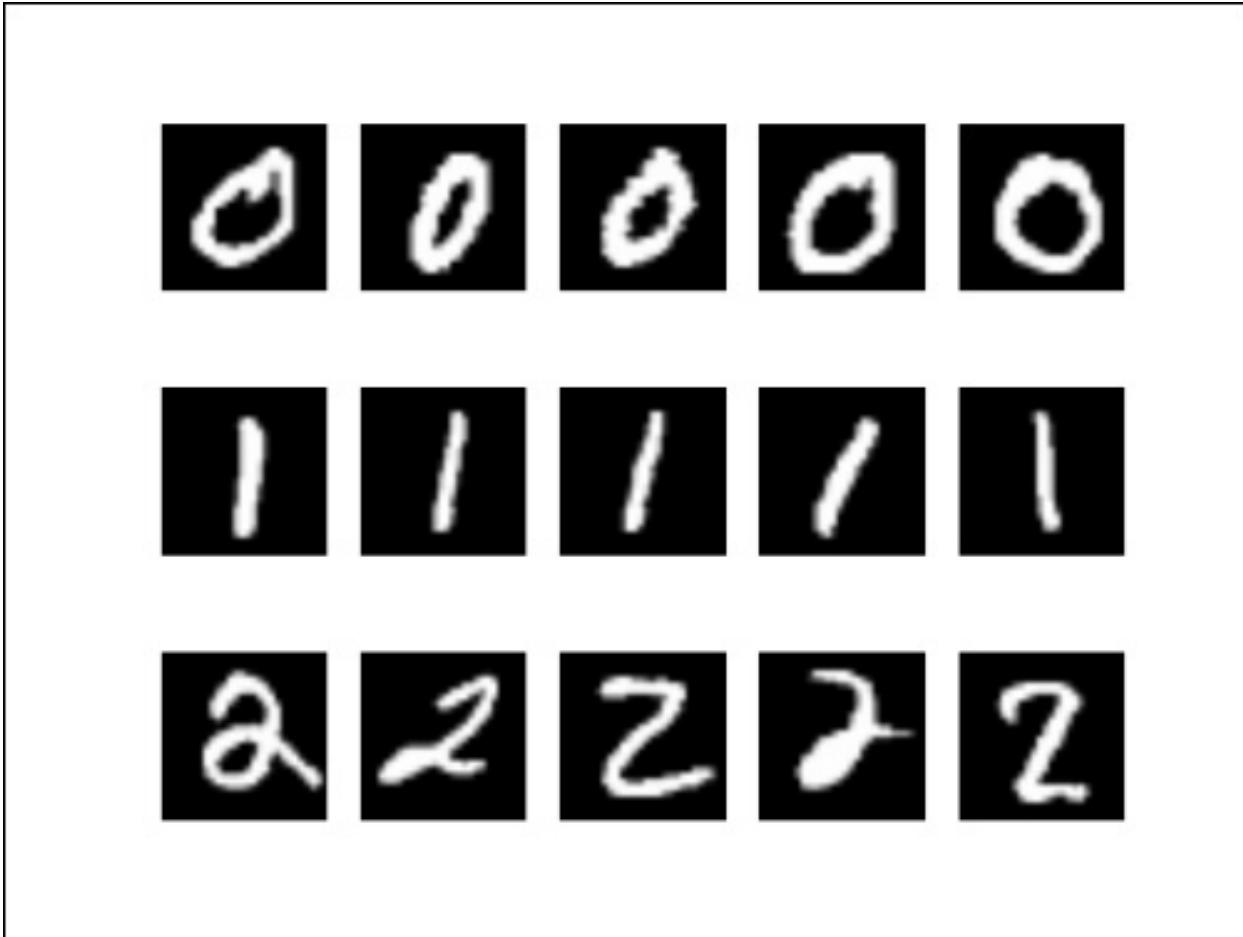
The Mixed National Institute of Standards and Technology database is a collection of 70,000 images of handwritten digits. The digits were sampled from documents written by employees of the US Census Bureau and American high school students. The images are grayscale and 28 x 28 pixels in dimension. Let's inspect some of the images using the following script:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.datasets import fetch_mldata
>>> import matplotlib.cm as cm

>>> digits = fetch_mldata('MNIST original',
data_home='data/mnist').data
>>> counter = 1
>>> for i in range(1, 4):
>>>     for j in range(1, 6):
>>>         plt.subplot(3, 5, counter)
>>>         plt.imshow(digits[(i - 1) * 8000 +
j].reshape((28, 28)), cmap=cm.Greys_r)
>>>         plt.axis('off')
>>>         counter += 1
>>> plt.show()
```

First, we load the data. scikit-learn provides the `fetch_mldata` convenience function to download the data set if it is not found on disk, and read it into an object. Then, we create a subplot for five instances for the digits

zero, one, and two. The script produces the following figure:



The MNIST data set is partitioned into a training set of 60,000 images and test set of 10,000 images. The dataset is commonly used to evaluate a variety of machine learning models; it is popular because little preprocessing is required. Let's use scikit-learn to build a classifier that can predict the digit depicted in an image.

First, we import the necessary classes:

```
from sklearn.datasets import fetch_mldata
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import scale
from sklearn.cross_validation import
train_test_split
from sklearn.svm import SVC
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import
classification_report
```

The script will fork additional processes during grid search, which requires execution from a `__main__` block.

```
if __name__ == '__main__':
    data = fetch_mldata('MNIST original',
data_home='data/mnist')
    X, y = data.data, data.target
    X = X/255.0**2 - 1
```

Next, we load the data using the `fetch_mldata` convenience function. We scale the features and center each feature around the origin. We then split the preprocessed data into training and test sets using the following line of code:

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

Next, we instantiate an `SVC`, or support vector classifier,

object. This object exposes an API like that of scikit-learn's other estimators; the classifier is trained using the `fit` method, and predictions are made using the `predict` method. If you consult the documentation for `SVC`, you will find that the estimator requires more hyperparameters than most of the other estimators we discussed. It is common for more powerful estimators to require more hyperparameters. The most interesting hyperparameters for `SVC` are set by the `kernel`, `gamma`, and `C` keyword arguments. The `kernel` keyword argument specifies the kernel to be used. scikit-learn provides implementations of the linear, polynomial, sigmoid, and radial basis function kernels. The `degree` keyword argument should also be set when the polynomial kernel is used. `C` controls regularization; it is similar to the `lambda` hyperparameter we used for logistic regression. The keyword argument `gamma` is the kernel coefficient for the sigmoid, polynomial, and RBF kernels. Setting these hyperparameters can be challenging, so we tune them by grid searching with the following code.

```
pipeline = Pipeline([
    ('clf', SVC(kernel='rbf', gamma=0.01,
C=100))
])
print X_train.shape
parameters = {
    'clf__gamma': (0.01, 0.03, 0.1, 0.3, 1),
```

```

        'clf__C': (0.1, 0.3, 1, 3, 10, 30),
    }
grid_search = GridSearchCV(pipeline,
parameters, n_jobs=2, verbose=1,
scoring='accuracy')
grid_search.fit(X_train[:10000],
y_train[:10000])
print 'Best score: %0.3f' %
grid_search.best_score_
print 'Best parameters set:'
best_parameters =
grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print '\t%s: %r' % (param_name,
best_parameters[param_name])
predictions = grid_search.predict(X_test)
print classification_report(y_test,
predictions)

```

The following is the output of the preceding script:

```

Fitting 3 folds for each of 30 candidates,
totalling 90 fits
[Parallel(n_jobs=2)]: Done   1 jobs          |
elapsed:  7.7min
[Parallel(n_jobs=2)]: Done   50 jobs          |
elapsed: 201.2min
[Parallel(n_jobs=2)]: Done  88 out of  90 |
elapsed: 304.8min remaining:  6.9min
[Parallel(n_jobs=2)]: Done  90 out of  90 |
elapsed: 309.2min finished
Best score: 0.966
Best parameters set:

```

	clf_C: 3	clf_gamma: 0.01	precision	recall	f1-score
support					
1758	0.0	0.98	0.99	0.99	
1968	1.0	0.98	0.99	0.98	
1727	2.0	0.95	0.97	0.96	
1803	3.0	0.97	0.95	0.96	
1714	4.0	0.97	0.98	0.97	
1535	5.0	0.96	0.96	0.96	
1758	6.0	0.98	0.98	0.98	
1840	7.0	0.97	0.96	0.97	
1668	8.0	0.95	0.96	0.96	
1729	9.0	0.96	0.95	0.96	
avg / total		0.97	0.97	0.97	
17500					

The best model has an average F1 score of 0.97; this score can be increased further by training on more than the first ten thousand instances.

Classifying characters in natural images

Now let's try a more challenging problem. We will classify alphanumeric characters in natural images. The Chars74K dataset, collected by T. E. de Campos, B. R. Babu, and M. Varma for *Character Recognition in Natural Images*, contains more than 74,000 images of the digits zero through to nine and the characters for both cases of the English alphabet. The following are three examples of images of the lowercase letter *z*. Chars74K can be downloaded from

<http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/>.



Several types of images comprise the collection. We will use 7,705 images of characters that were extracted from photographs of street scenes taken in Bangalore, India. In contrast to MNIST, the images in this portion of Chars74K depict the characters in a variety of fonts, colors, and perturbations. After expanding the archive, we will use the files in the English/Img/GoodImg/Bmp/ directory. First we will import the necessary classes.

```
import os
import numpy as np
from sklearn.svm import SVC
from sklearn.cross_validation import
train_test_split
from sklearn.metrics import
classification_report
import Image
```

Next we will define a function that resizes images using the Python Image Library:

```
def resize_and_crop(image, size):
    img_ratio = image.size[0] /
float(image.size[1])
    ratio = size[0] / float(size[1])
    if ratio > img_ratio:
        image = image.resize((size[0], size[0] *
image.size[1] / image.size[0]), Image.ANTIALIAS)
        image = image.crop((0, 0, 30, 30))
    elif ratio < img_ratio:
        image = image.resize((size[1] *
```

```

image.size[0] / image.size[1], size[1]),
Image.ANTIALIAS)
    image = image.crop((0, 0, 30, 30))
else:
    image = image.resize((size[0], size[1]),
Image.ANTIALIAS)
return image

```

Then we load will the images for each of the 62 classes and convert them to grayscale. Unlike MNIST, the images of Chars74K do not have consistent dimensions, so we will resize them to 30 pixels on a side using the `resize_and_crop` function we defined. Finally, we will convert the processed images to a NumPy array:

```

X = []
y = []

for path, subdirs, files in
os.walk('data/English/Img/GoodImg/Bmp/'):
    for filename in files:
        f = os.path.join(path, filename)
        img = Image.open(f).convert('L') #
convert to grayscale
        img_resized = resize_and_crop(img, (30,
30))
        img_resized =
np.asarray(img_resized.getdata(),
dtype=np.float64) \
            .reshape((img_resized.size[1] *
img_resized.size[0], 1))
        target = filename[3:filename.index('-')]

```

```

X.append(img_resized)
y.append(target)

X = np.array(X)
X = X.reshape(X.shape[:2])

```

```

We will then train a support vector classifier
with a polynomial kernel.
classifier =
SVC(verbose=0, kernel='poly', degree=3)
X_train, X_test, y_train, y_test =
train_test_split(X, y, random_state=1)
classifier.fit(X_train, y_train)
predictions = classifier.predict(X_test)
print classification_report(y_test, predictions)

```

The preceding script produces the following output:

		precision	recall	f1-score
	support			
23	001	0.24	0.22	0.23
20	002	0.24	0.45	0.32
	...			
13	061	0.33	0.15	0.21
8	062	0.08	0.25	0.12
1927	avg / total	0.41	0.34	0.36

It is apparent that this is a more challenging task than classifying digits in MNIST. The appearances of the characters vary more widely, the characters are perturbed more since the images were sampled from photographs rather than scanned documents. Furthermore, there are far fewer training instances for each class in Chars74K than there are in MNIST. The performance of the classifier could be improved by adding training data, preprocessing the images differently, or using more sophisticated feature representations.

Summary

In this chapter, we discussed the support vector machine —a powerful model that can mitigate some of the limitations of perceptrons. The perceptron can be used effectively for linearly separable classification problems, but it cannot express more complex decision boundaries without expanding the feature space to higher dimensions. Unfortunately, this expansion is prone to computation and generalization problems. Support vector machines redress the first problem using kernels, which avoid explicitly computing the feature mapping. They redress the second problem by maximizing the margin between the decision boundary and the nearest instances. In the next chapter, we will discuss models called artificial neural networks, which, like support vector machines, extend the perceptron to overcome its limitations.

Chapter 10. From the Perceptron to Artificial Neural Networks

In [Chapter 8](#), *The Perceptron*, we introduced the perceptron, which is a linear model for binary classification. You learned that the perceptron is not a universal function approximator; its decision boundary must be a hyperplane. In the previous chapter we introduced the support vector machine, which redresses some of the perceptron's limitations by using kernels to efficiently map the feature representations to a higher dimensional space in which the instances are linearly separable. In this chapter, we will discuss **artificial neural networks**, which are powerful nonlinear models for classification and regression that use a different strategy to overcome the perceptron's limitations.

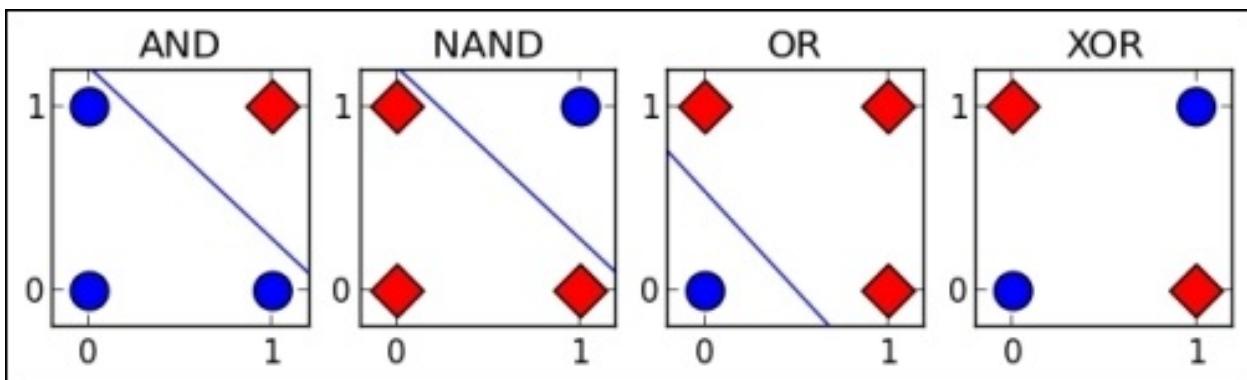
If the perceptron is analogous to a neuron, an artificial neural network, or **neural net**, is analogous to a brain. As billions of neurons with trillions of synapses comprise a human brain, an artificial neural network is a directed graph of perceptrons or other artificial neurons. The graph's edges are weighted; these weights are the

parameters of the model that must be learned.

Entire books describe individual aspects of artificial neural networks; this chapter will provide an overview of their structure and training. At the time of writing, some artificial neural networks have been developed for scikit-learn, but they are not available in Version 0.15.2. Readers can follow the examples in this chapter by checking out a fork of scikit-learn 0.15.1 that includes the neural network module. The implementations in this fork are likely to be merged into future versions of scikit-learn without any changes to the API described in this chapter.

Nonlinear decision boundaries

Recall from [Chapter 8, *The Perceptron*](#), that while some Boolean functions such as AND, OR, and NAND can be approximated by the perceptron, the linearly inseparable function XOR cannot, as shown in the following plots:



Let's review XOR in more detail to develop an intuition for the power of artificial neural networks. In contrast to AND, which outputs 1 when both of its inputs are equal to 1, and OR, which outputs 1 when at least one of the inputs are equal to 1, the output of XOR is 1 when exactly one of its inputs are equal to 1. We could view XOR as outputting 1 when two conditions are true. The first condition is that at least one of the inputs must be equal to

1; this is the same condition that OR tests. The second condition is that not both of the inputs are equal to 1; NAND tests this condition. We can produce the same output as XOR by processing the input with both OR and NAND and then verifying that the outputs of both functions are equal to 1 using AND. That is, the functions OR, NAND, and AND can be composed to produce the same output as XOR.

The following tables provide the truth tables for XOR, OR, AND, and NAND for the inputs A and B . From these tables we can verify that inputting the output of OR and NAND to AND produces the same output as inputting A and B to XOR:

A	B	A AND B	A NAND B	A OR B	A XOR B
0	0	0	1	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	0	1	0

A	B	A OR B	A NAND B	(A OR B) AND (A NAND B)
0	0	0	1	0

0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

Instead of trying to represent XOR with a single perceptron, we will build an artificial neural network from multiple artificial neurons that each approximate a linear function. Each instance's feature representation will be input to two neurons; one neuron will represent NAND and the other will represent OR. The output of these neurons will be received by a third neuron that represents AND to test whether both of XOR's conditions are true.

Feedforward and feedback artificial neural networks

Artificial neural networks are described by three components. The first is the model's **architecture**, or topology, which describes the layers of neurons and structure of the connections between them. The second component is the activation function used by the artificial neurons. The third component is the learning algorithm that finds the optimal values of the weights.

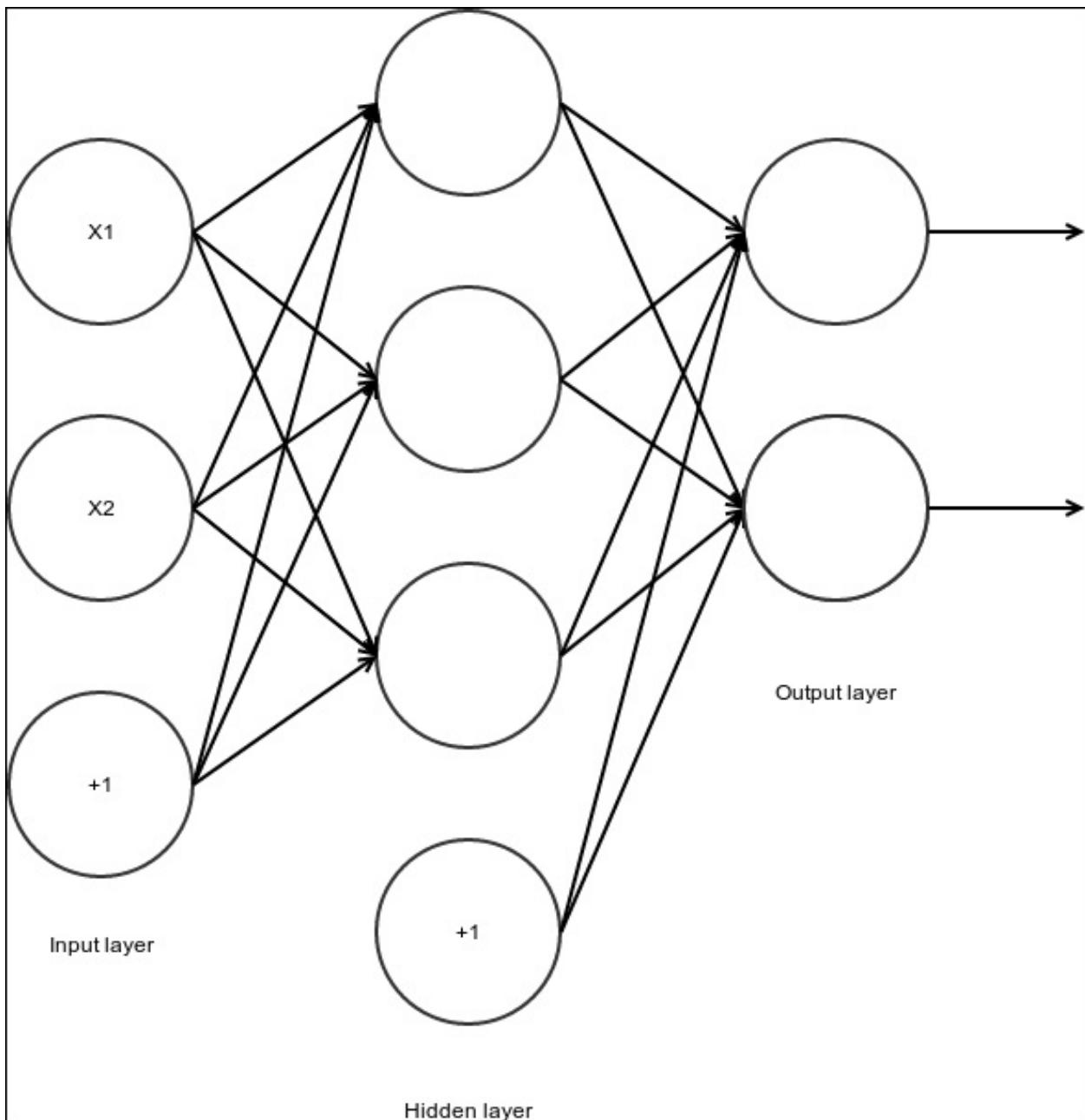
There are two main types of artificial neural networks. **Feedforward neural networks** are the most common type of neural net, and are defined by their directed acyclic graphs. Signals only travel in one direction—towards the output layer—in feedforward neural networks. Conversely, **feedback neural networks**, or recurrent neural networks, do contain cycles. The feedback cycles can represent an internal state for the network that can cause the network's behavior to change over time based on its input. Feedforward neural networks are commonly used to learn a function to map an input to an output. The temporal behavior of feedback neural networks makes them suitable for processing sequences of inputs. Because feedback neural networks are not

implemented in scikit-learn, we will limit our discussion to only feedforward neural networks.

Multilayer perceptrons

The **multilayer perceptron (MLP)** is the one of the most commonly used artificial neural networks. The name is a slight misnomer; a multilayer perceptron is not a single perceptron with multiple layers, but rather multiple layers of artificial neurons that can be perceptrons. The layers of the MLP form a directed, acyclic graph. Generally, each layer is fully connected to the subsequent layer; the output of each artificial neuron in a layer is an input to every artificial neuron in the next layer towards the output. MLPs have three or more layers of artificial neurons.

The **input layer** consists of simple input neurons. The input neurons are connected to at least one **hidden layer** of artificial neurons. The hidden layer represents latent variables; the input and output of this layer cannot be observed in the training data. Finally, the last hidden layer is connected to an **output layer**. The following diagram depicts the architecture of a multilayer perceptron with three layers. The neurons labeled **+1** are bias neurons and are not depicted in most architecture diagrams.



The artificial neurons, or **units**, in the hidden layer commonly use nonlinear activation functions such as the hyperbolic tangent function and the logistic function,

which are given by the following equations:

$$f(x) = \tanh(x)$$

$$f(x) = \frac{1}{1+e^{-x}}$$

As with other supervised models, our goal is to find the values of the weights that minimize the value of a cost function. The mean squared error cost function is commonly used with multilayer perceptrons. It is given by the following equation, where m is the number of training instances:

$$MSE = \frac{1}{m} \sum_{i=1}^m m(y_i - f(x_i))^2$$

Minimizing the cost function

The **backpropagation** algorithm is commonly used in conjunction with an optimization algorithm such as gradient descent to minimize the value of the cost function. The algorithm takes its name from a portmanteau of *backward propagation*, and refers to the direction in which errors flow through the layers of the network. Backpropagation can theoretically be used to train a feedforward network with any number of hidden units arranged in any number of layers, though computational power constrains this capability.

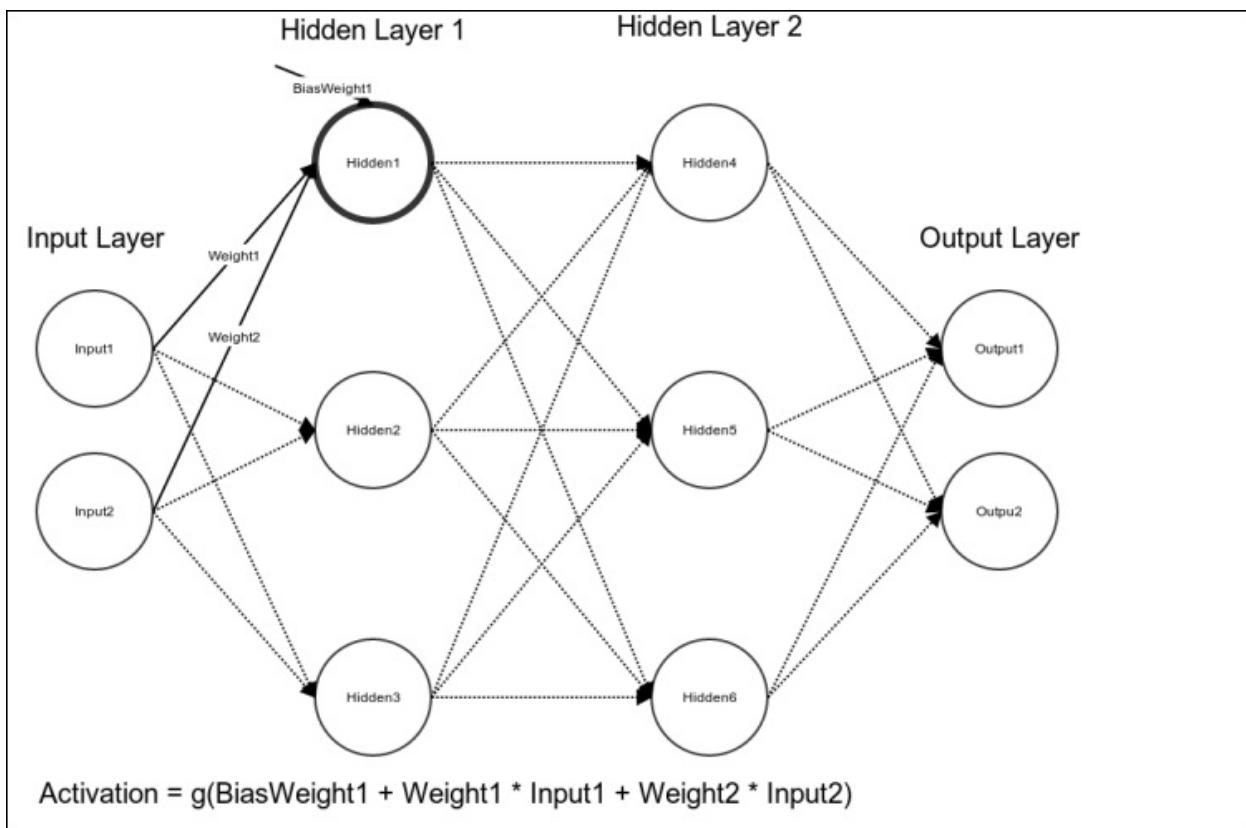
Backpropagation is similar to gradient descent in that it uses the gradient of the cost function to update the values of the model parameters. Unlike the linear models we have previously seen, neural nets contain hidden units that represent latent variables; we can't tell what the hidden units should do from the training data. If we do not know what the hidden units should do, we cannot calculate their errors and we cannot calculate the gradient of cost function with respect to their weights. A naive solution to overcome this is to randomly perturb the weights for the hidden units. If a random change to one of the weights decreases the value of the cost function, we save the change and randomly change the value of another weight.

An obvious problem with this solution is its prohibitive computational cost. Backpropagation provides a more efficient solution.

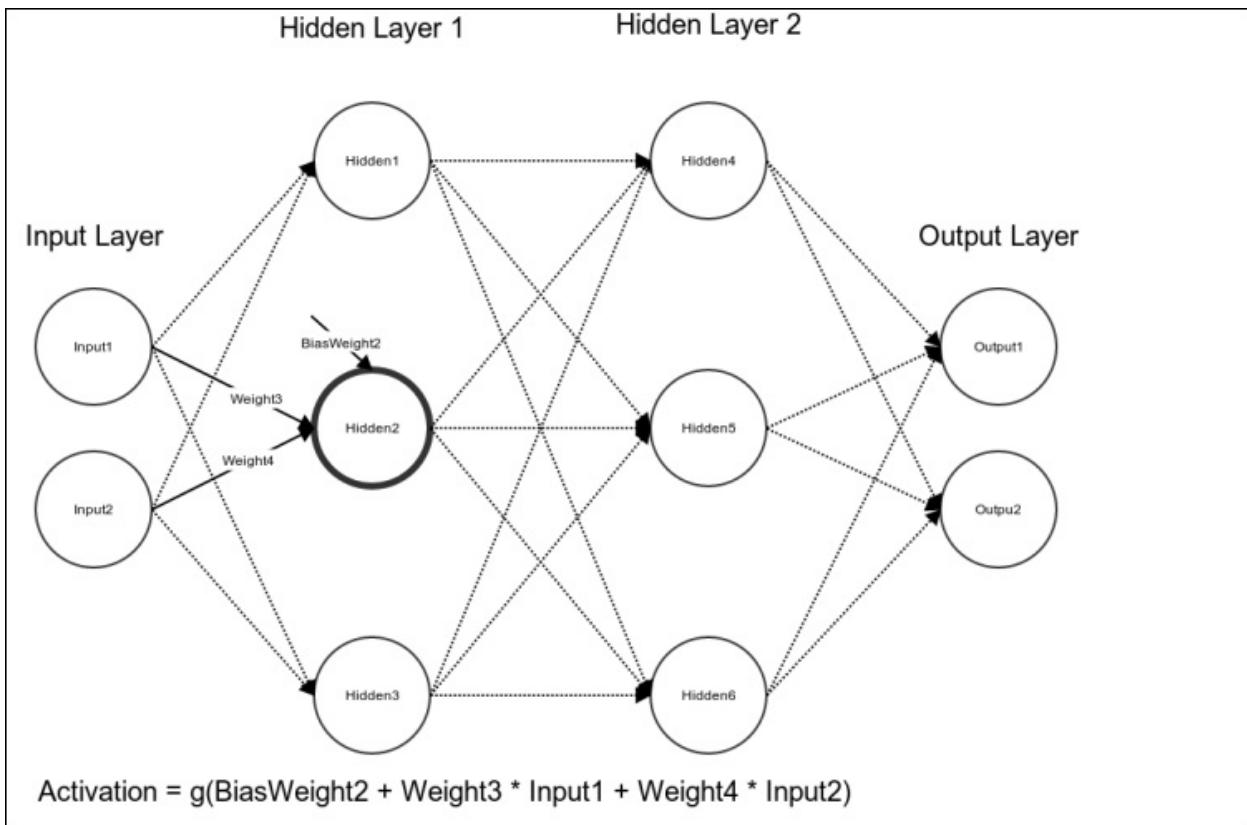
We will step through training a feedforward neural network using backpropagation. This network has two input units, two hidden layers that both have three hidden units, and two output units. The input units are both fully connected to the first hidden layer's units, called `Hidden1`, `Hidden2`, and `Hidden3`. The edges connecting the units are initialized to small random weights.

Forward propagation

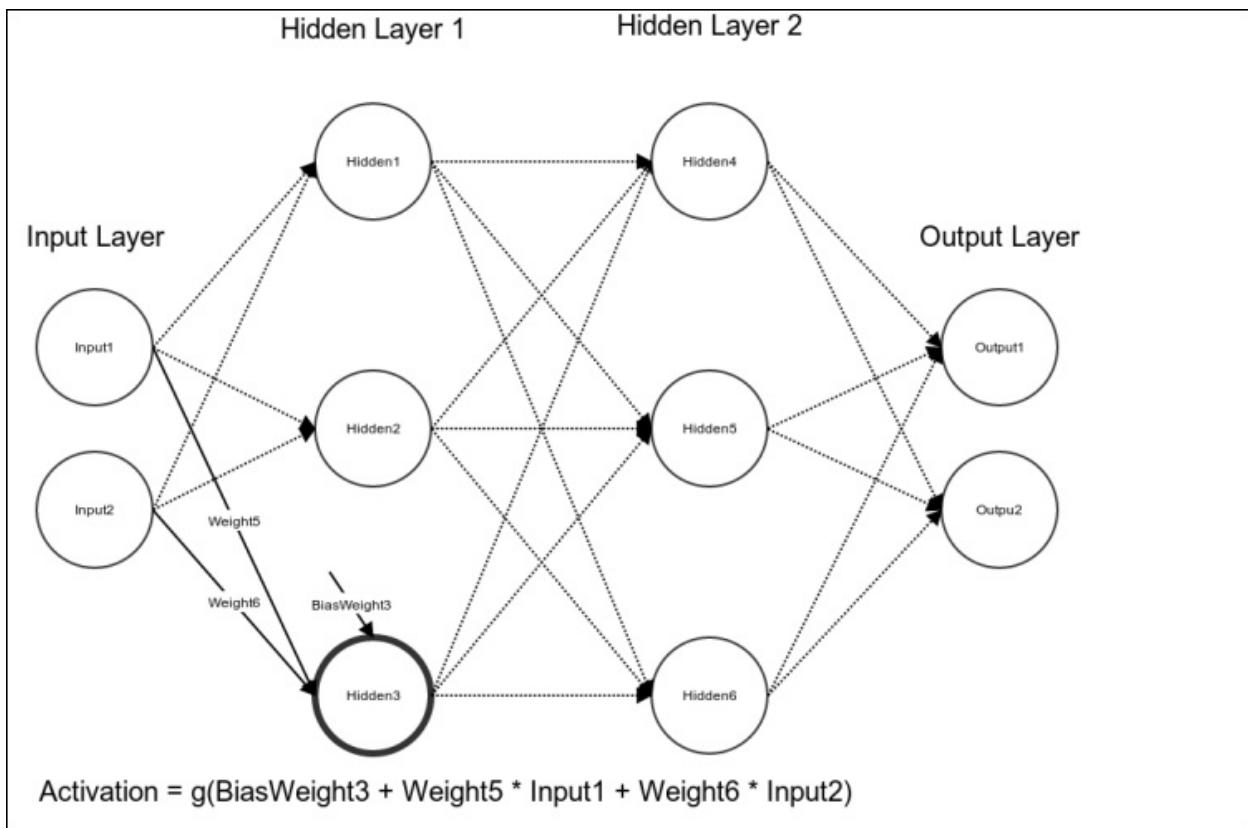
During the forward propagation stage, the features are input to the network and fed through the subsequent layers to produce the output activations. First, we compute the activation for the unit `Hidden1`. We find the weighted sum of input to `Hidden1`, and then process the sum with the activation function. Note that `Hidden1` receives a constant input from a bias unit that is not depicted in the diagram in addition to the inputs from the input units. In the following diagram, $g(x)$ is the activation function:



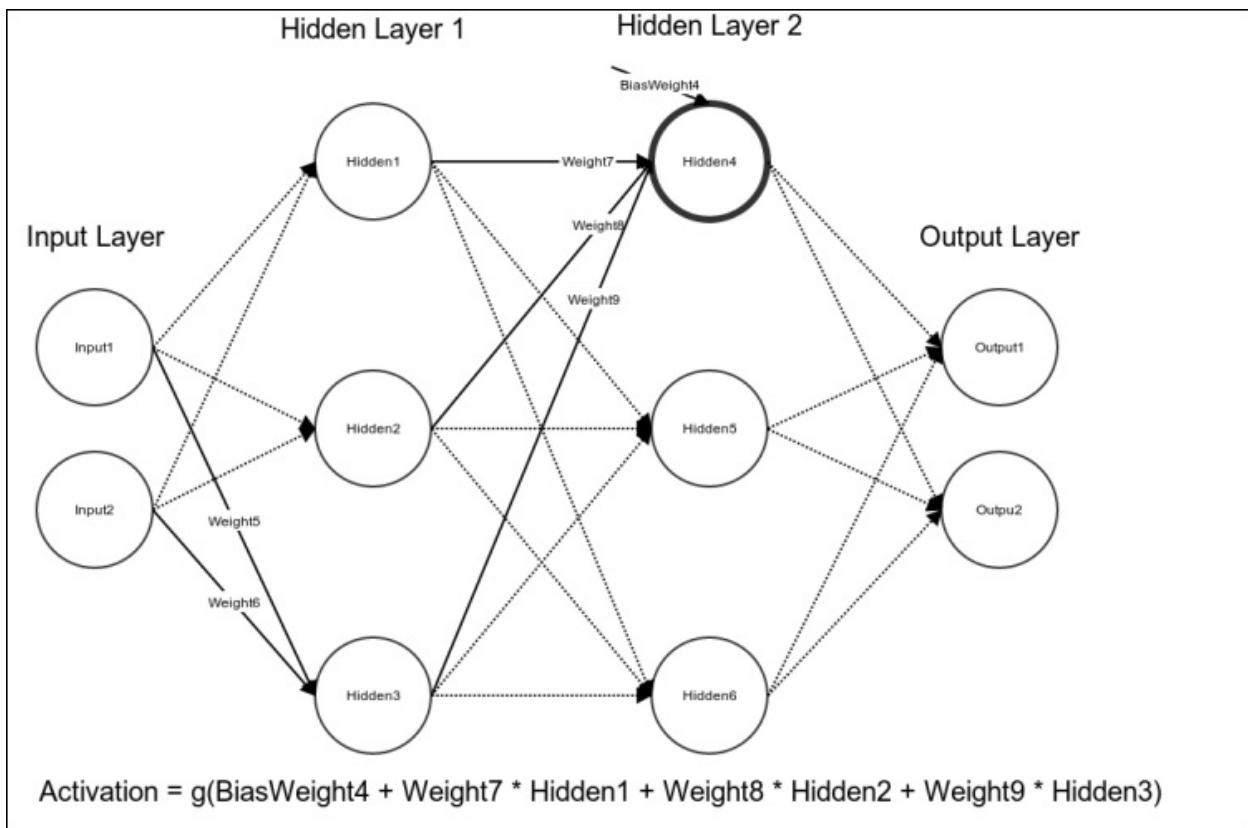
Next, we compute the activation for the second hidden unit. Like the first hidden unit, it receives weighted inputs from both of the input units and a constant input from a bias unit. We then process the weighted sum of the inputs, or **preactivation**, with the activation function as shown in the following figure:



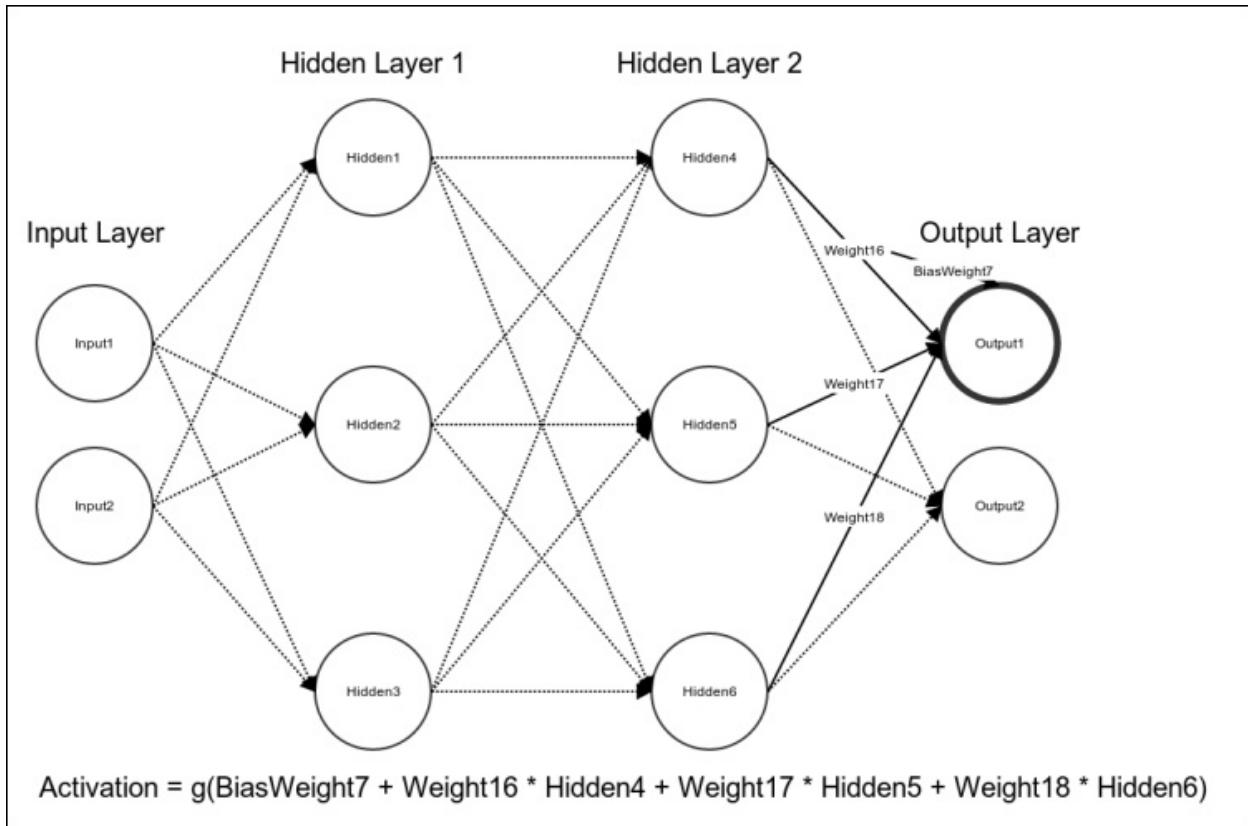
We then compute the activation for **Hidden3** in the same manner:



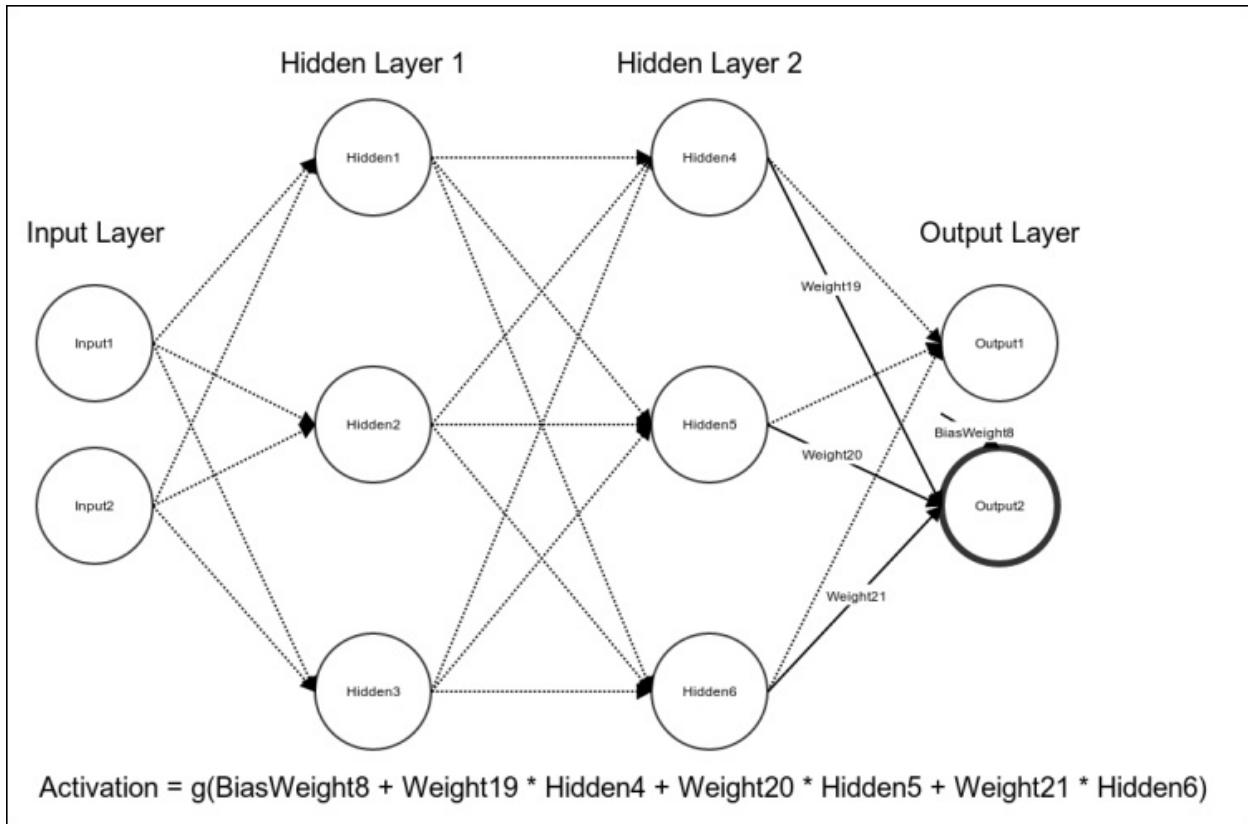
Having computed the activations of all of the hidden units in the first layer, we proceed to the second hidden layer. In this network, the first hidden layer is fully connected to the second hidden layer. Similar to the units in the first hidden layer, the units in the second hidden layer receive a constant input from bias units that are not depicted in the diagram. We proceed to compute the activation of Hidden4:



We next compute the activations of `Hidden5` and `Hidden6`. Having computed the activations of all of the hidden units in the second hidden layer, we proceed to the output layer in the following figure. The activation of `Output1` is the weighted sum of the second hidden layer's activations processed through an activation function. Similar to the hidden units, the output units both receive a constant input from a bias unit:



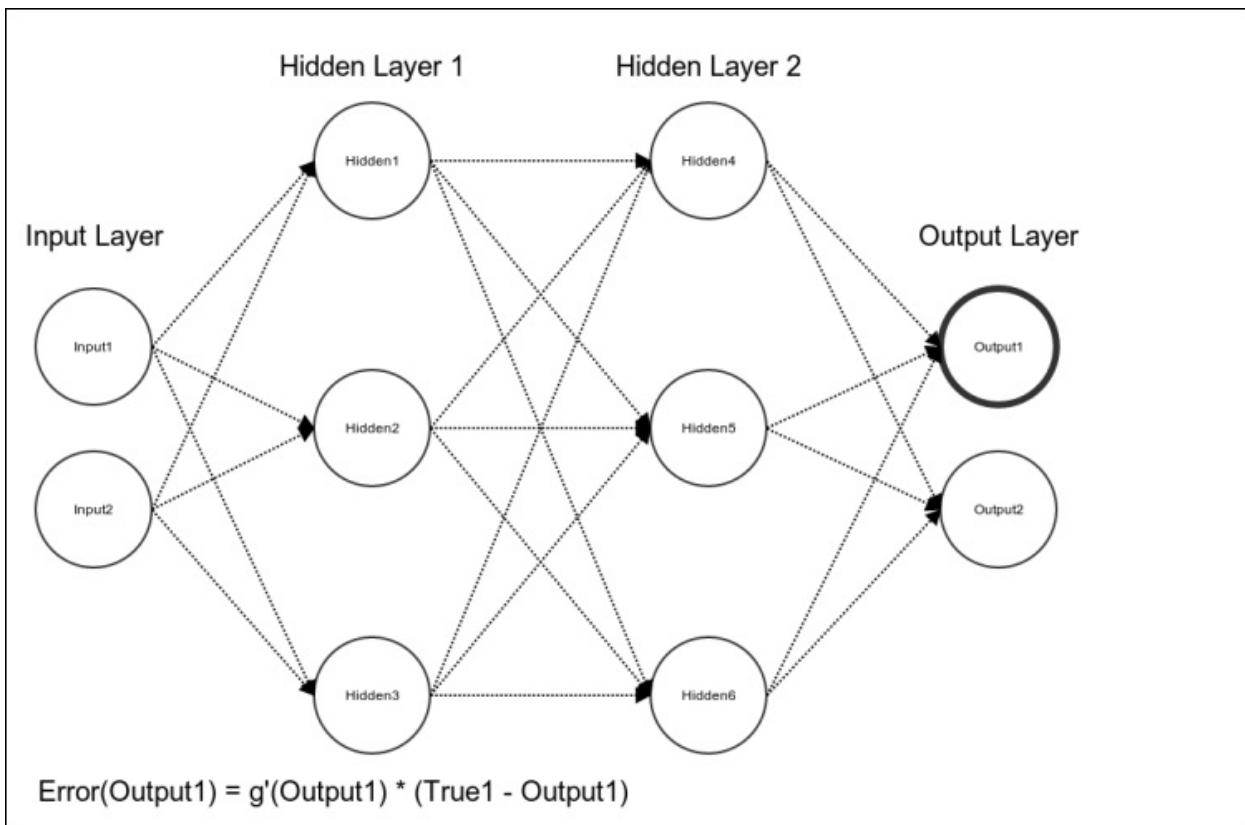
We calculate the activation of **Output2** in the same manner:



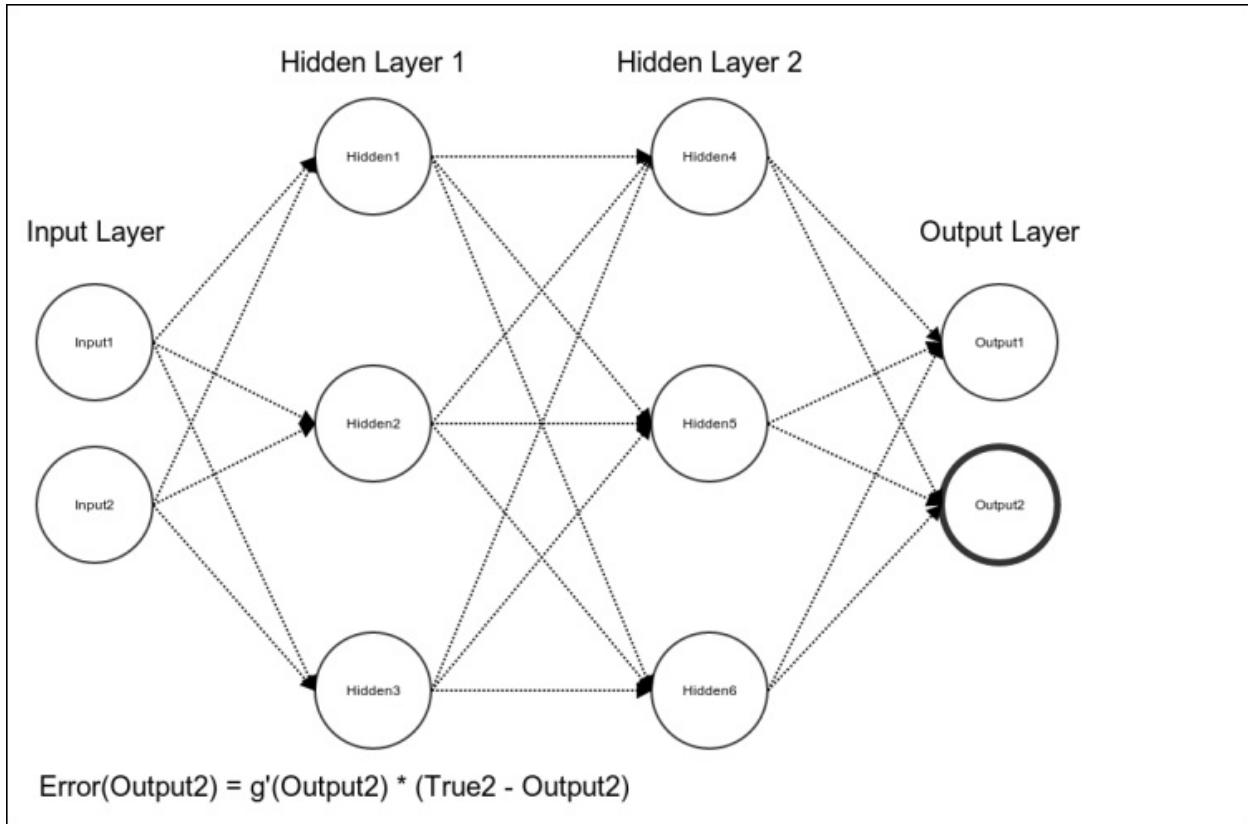
We computed the activations of all of the units in the network, and we have now completed forward propagation. The network is not likely to approximate the true function well using the initial random values of the weights. We must now update the values of the weights so that the network can better approximate our function.

Backpropagation

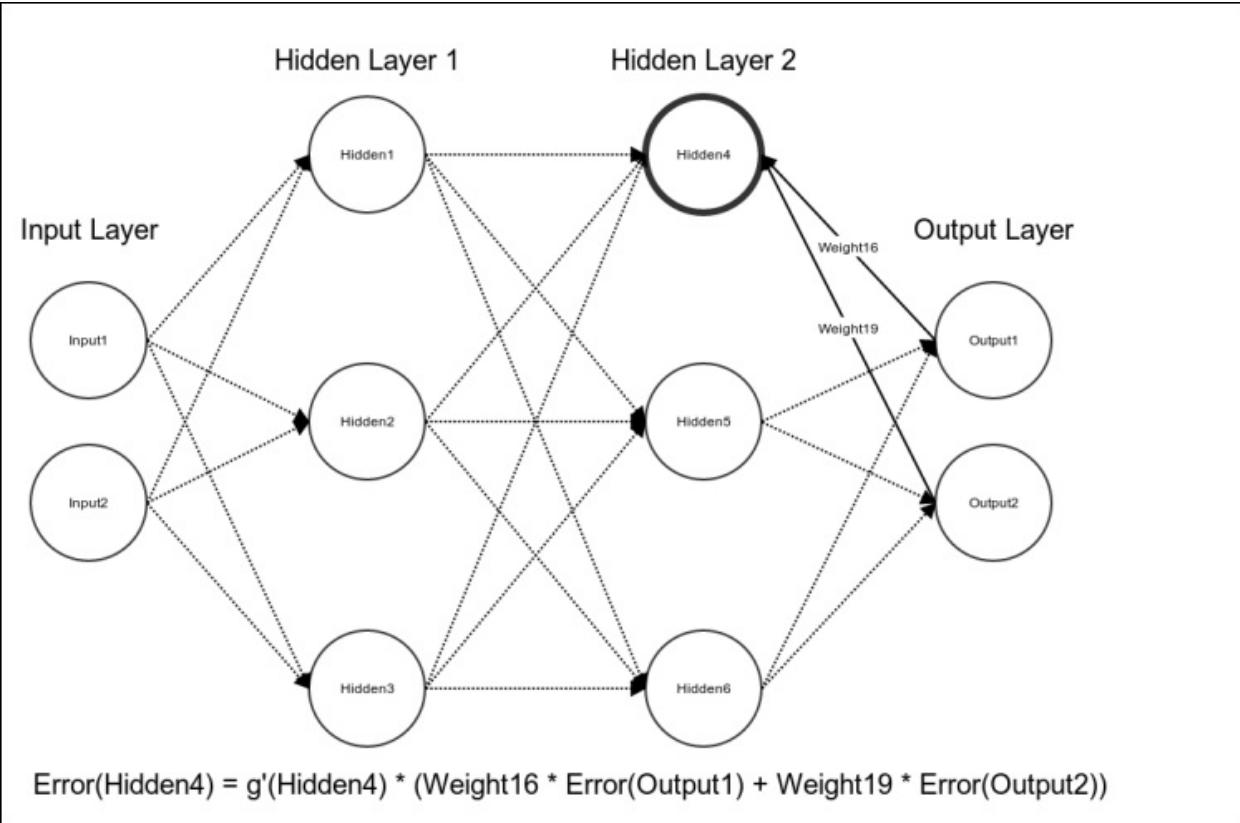
We can calculate the error of the network only at the output units. The hidden units represent latent variables; we cannot observe their true values in the training data and thus, we have nothing to compute their error against. In order to update their weights, we must propagate the network's errors backwards through its layers. We will begin with `Output1`. Its error is equal to the difference between the true and predicted outputs, multiplied by the partial derivative of the unit's activation:



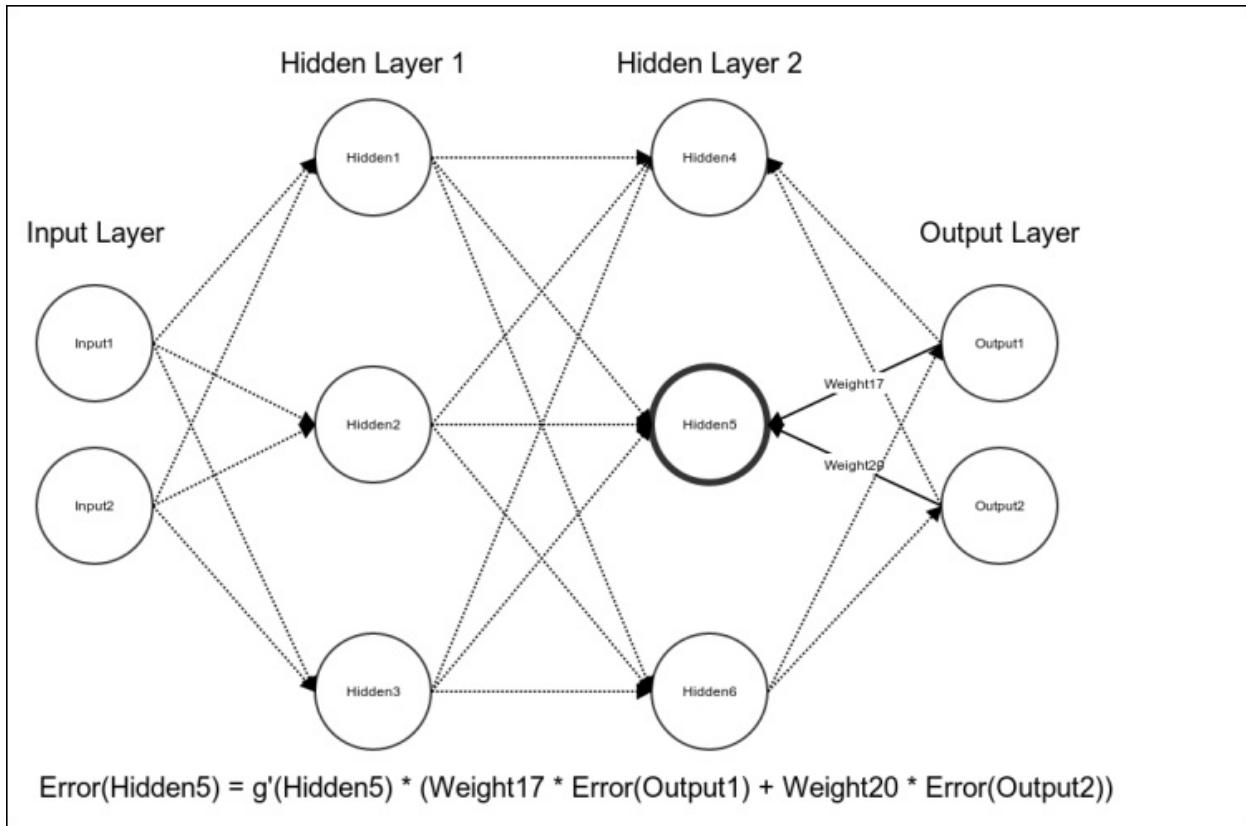
We then calculate the error of the second output unit:



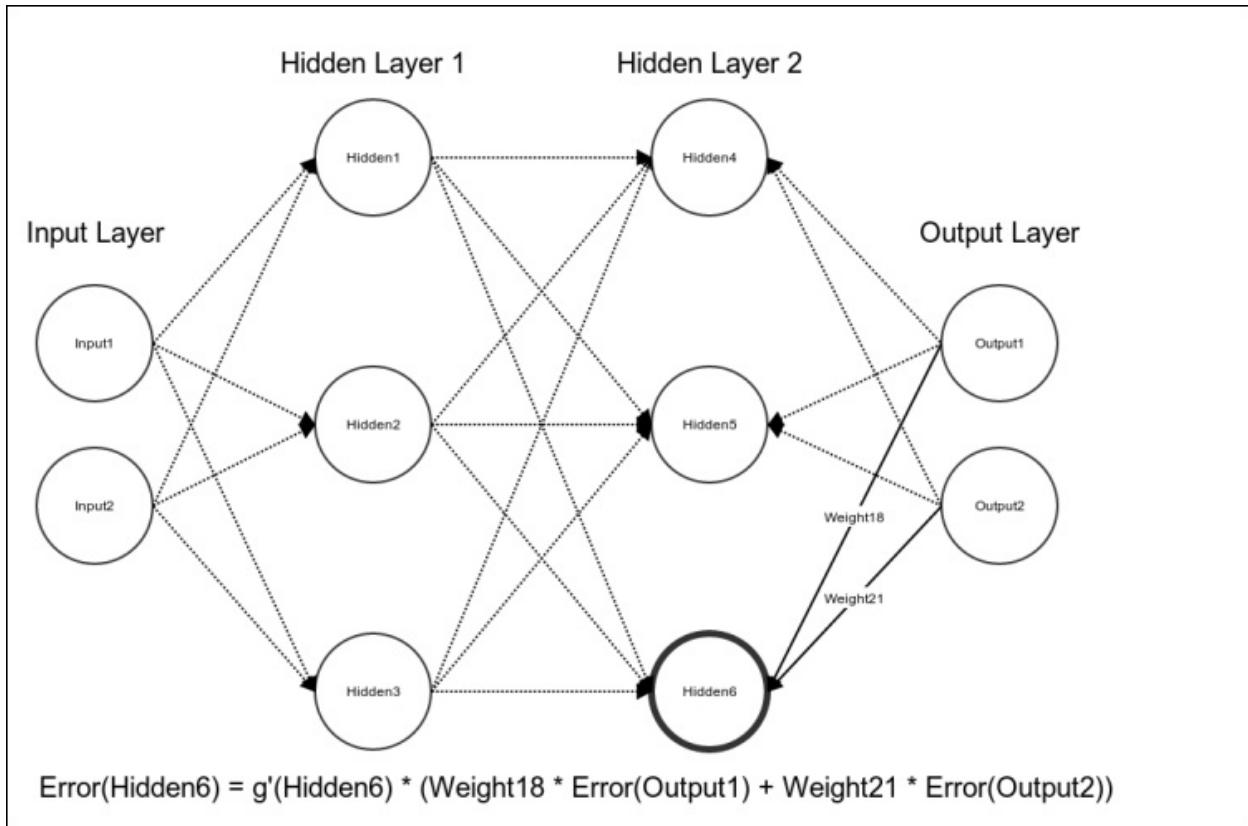
We computed the errors of the output layer. We can now propagate these errors backwards to the second hidden layer. First, we will compute the error of hidden unit `Hidden4`. We multiply the error of `Output1` by the value of the weight connecting `Hidden4` and `Output1`. We similarly weigh the error of `Output2`. We then add these errors and calculate the product of their sum and the partial derivative of `Hidden4`:



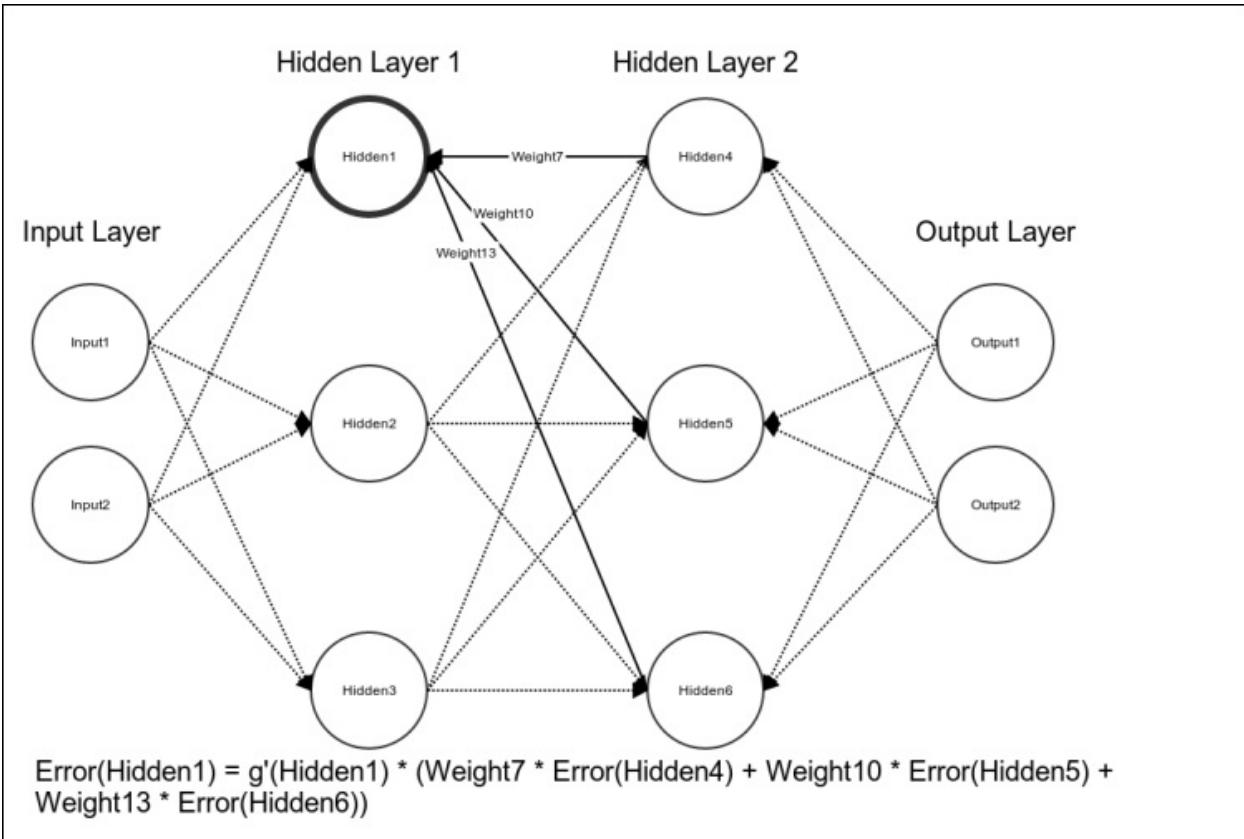
We similarly compute the errors of **Hidden5**:



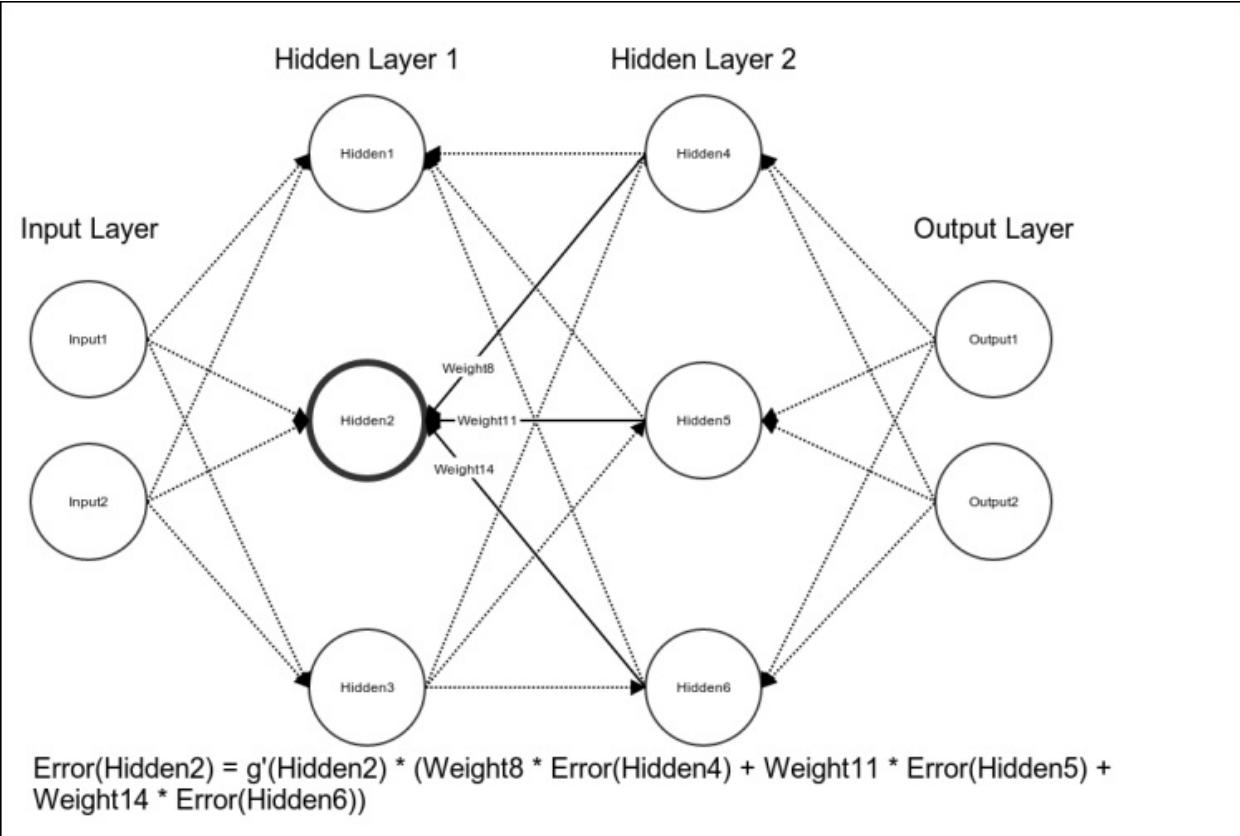
We then compute the **Hidden6** error in the following figure:



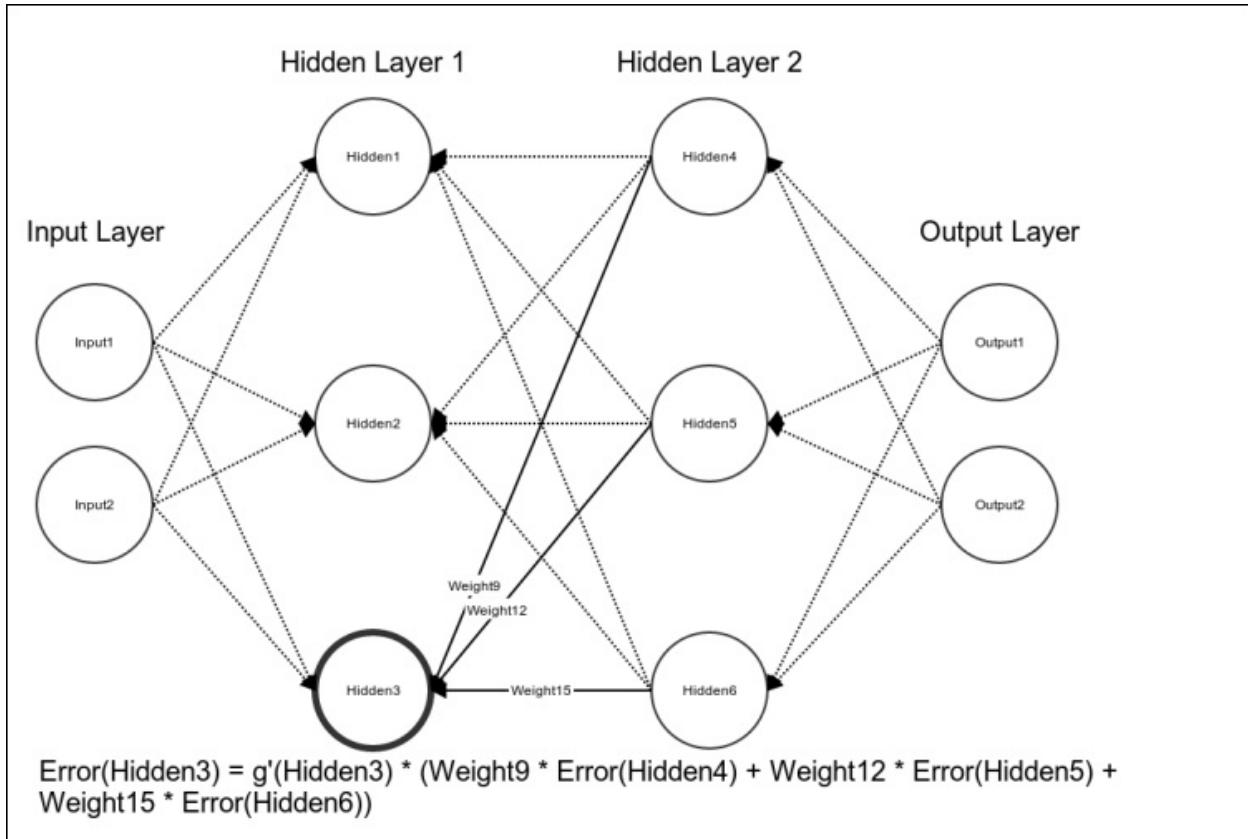
We calculated the error of the second hidden layer with respect to the output layer. Next, we will continue to propagate the errors backwards towards the input layer. The error of the hidden unit `Hidden1` is the product of its partial derivative and the weighted sums of the errors in the second hidden layer:



We similarly compute the error for hidden unit `Hidden2`:



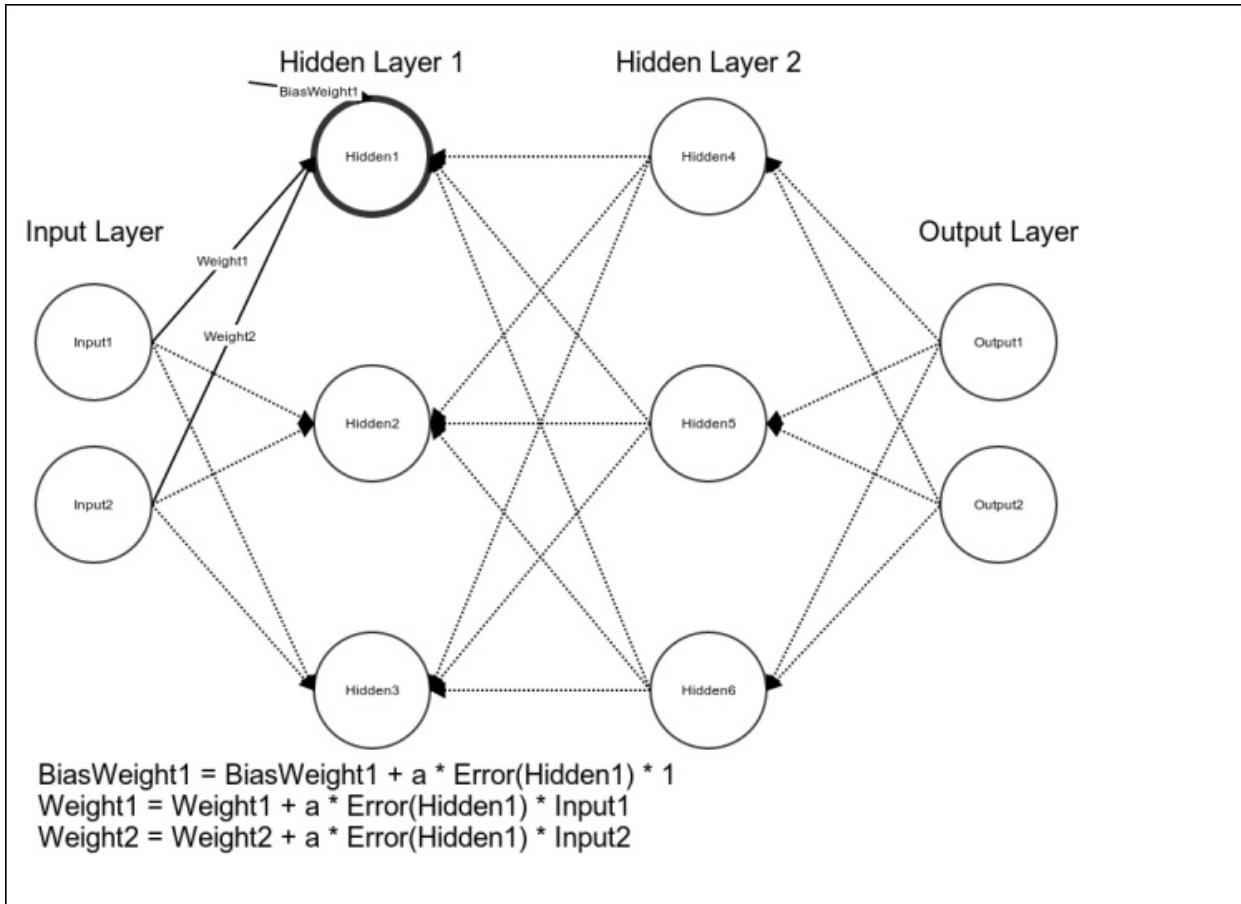
We similarly compute the error for `Hidden3`:



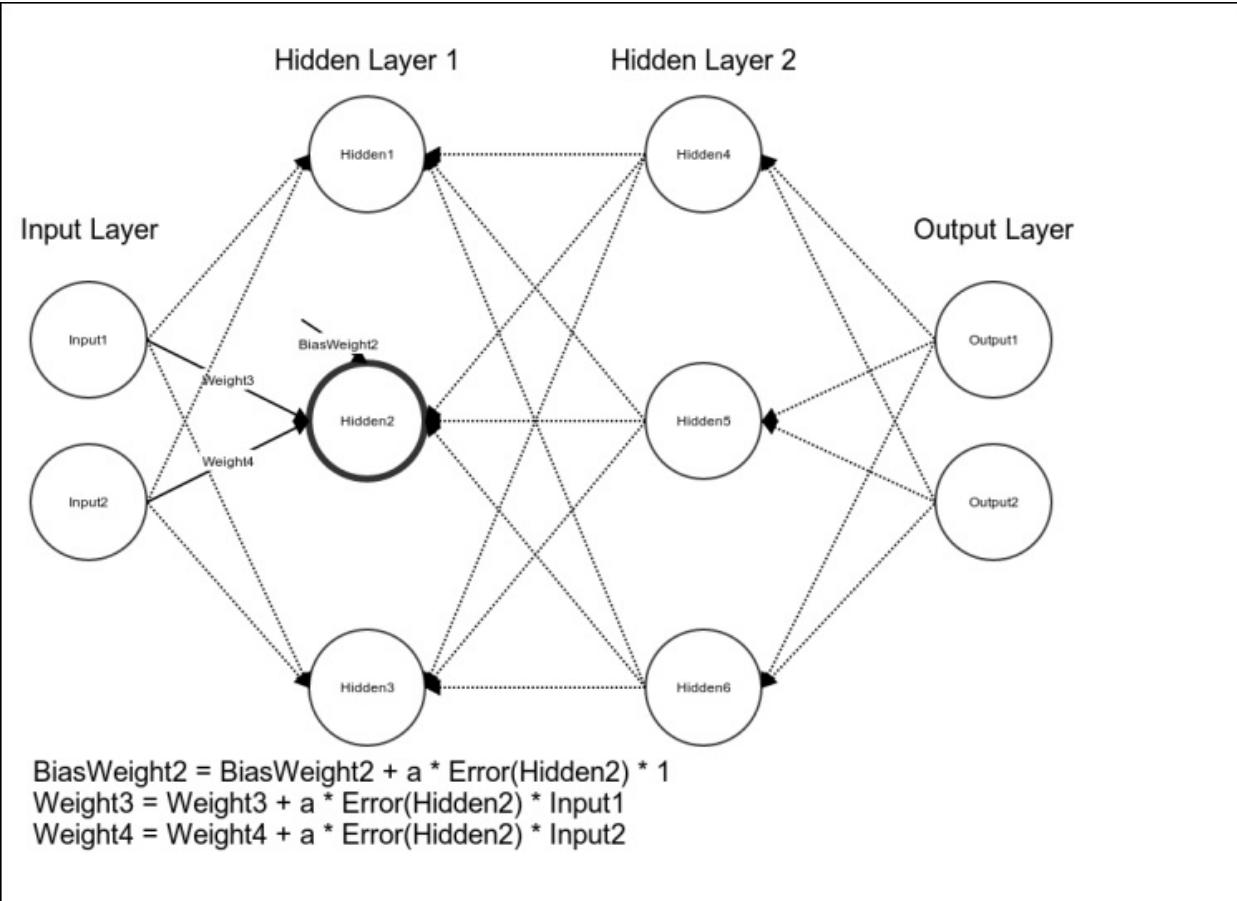
We computed the errors of the first hidden layer. We can now use these errors to update the values of the weights. We will first update the weights for the edges connecting the input units to `Hidden1` as well as the weight for the edge connecting the bias unit to `Hidden1`. We will increment the value of the weight connecting `Input1` and `Hidden1` by the product of the learning rate, error of `Hidden1`, and the value of `Input1`.

We will similarly increment the value of `Weight2` by the

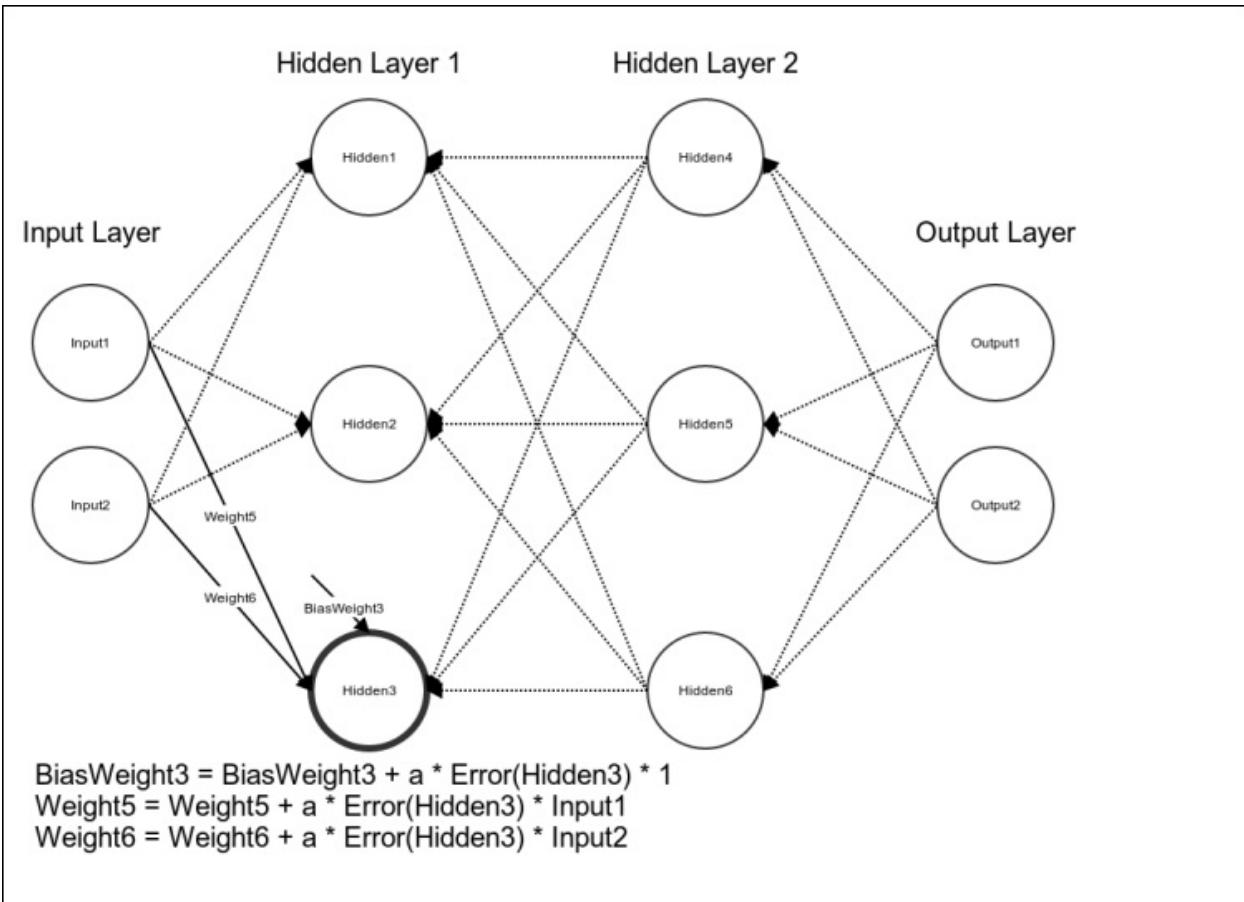
product of the learning rate, error of `Hidden1`, and the value of `Input2`. Finally, we will increment the value of the weight connecting the bias unit to `Hidden1` by the product of the learning rate, error of `Hidden1`, and one.



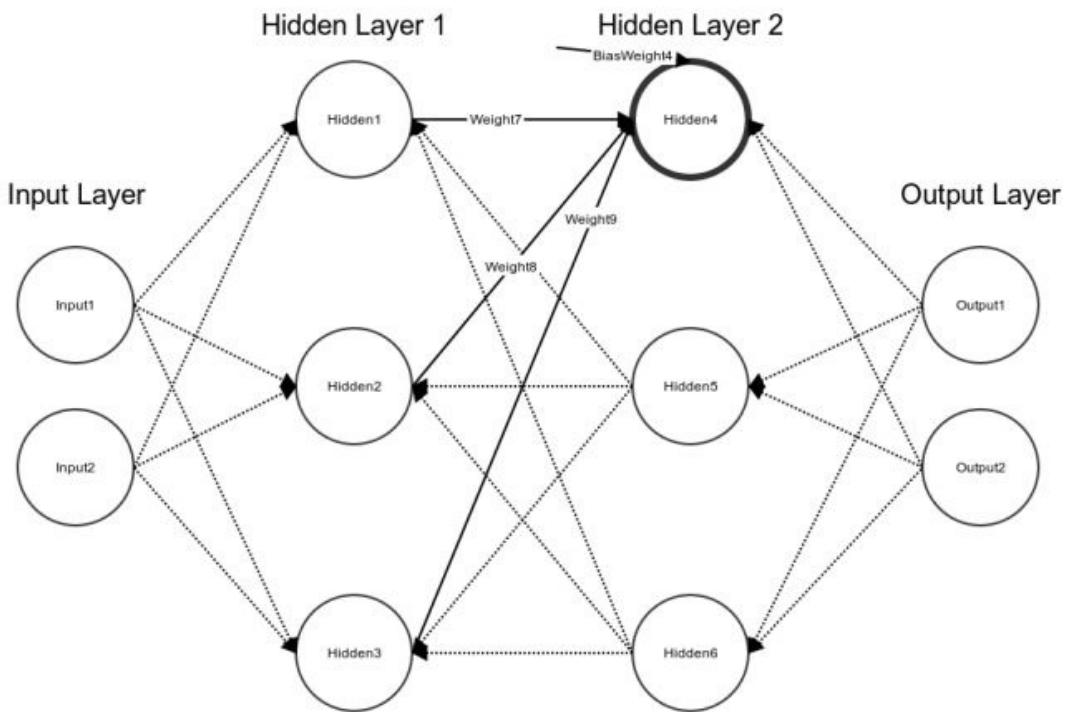
We will then update the values of the weights connecting hidden unit `Hidden2` to the input units and the bias unit using the same method:



Next, we will update the values of the weights connecting the input layer to **Hidden3**:

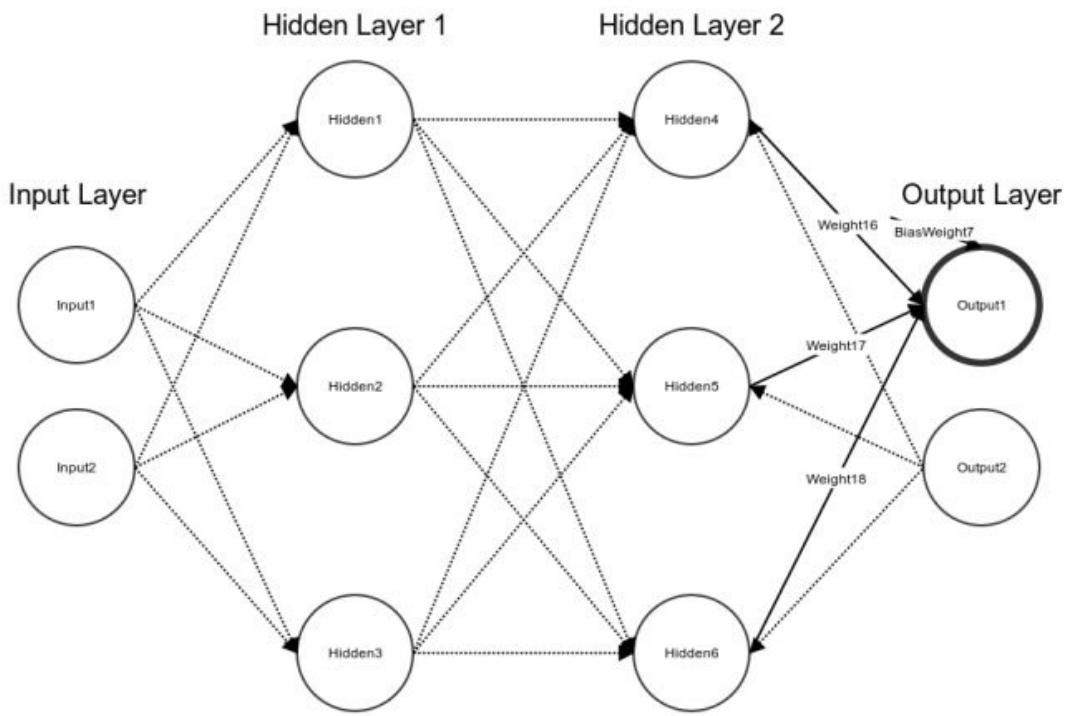


Since the values of the weights connecting the input layer to the first hidden layer is updated, we can continue to the weights connecting the first hidden layer to the second hidden layer. We will increment the value of `Weight7` by the product of the learning rate, error of `Hidden4`, and the output of `Hidden1`. We continue to similarly update the values of weights `Weight8` to `Weight15`:



$$\begin{aligned} \text{BiasWeight4} &= \text{BiasWeight4} + a * \text{Error}(\text{Hidden4}) * 1 \\ \text{Weight7} &= \text{Weight7} + a * \text{Error}(\text{Hidden4}) * \text{Hidden1} \\ \text{Weight8} &= \text{Weight8} + a * \text{Error}(\text{Hidden4}) * \text{Hidden2} \\ \text{Weight9} &= \text{Weight9} + a * \text{Error}(\text{Hidden4}) * \text{Hidden3} \end{aligned}$$

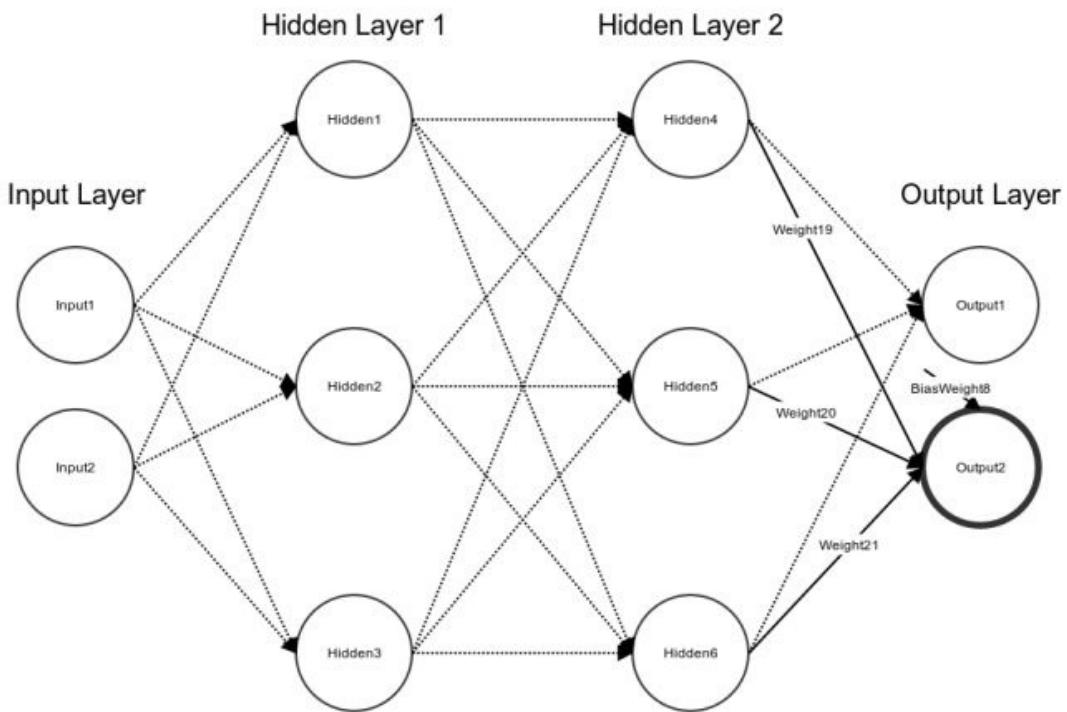
The weights for `Hidden5` and `Hidden6` are updated in the same way. We updated the values of the weights connecting the two hidden layers. We can now update the values of the weights connecting the second hidden layer and the output layer. We increment the values of weights w_{16} through w_{21} using the same method that we used for the weights in the previous layers:



```

BiasWeight7 = BiasWeight7 + a * Error(Output1) * 1
Weight16 = Weight16 + a * Error(Output1) * Hidden3
Weight17 = Weight17 + a * Error(Output1) * Hidden4
Weight18 = Weight18 + a * Error(Output1) * Hidden6

```



$$\begin{aligned}
 \text{BiasWeight8} &= \text{BiasWeight8} + a * \text{Error}(\text{Output2}) * 1 \\
 \text{Weight19} &= \text{Weight19} + a * \text{Error}(\text{Output2}) * \text{Hidden3} \\
 \text{Weight20} &= \text{Weight20} + a * \text{Error}(\text{Output2}) * \text{Hidden4} \\
 \text{Weight21} &= \text{Weight21} + a * \text{Error}(\text{Output2}) * \text{Hidden6}
 \end{aligned}$$

After incrementing the value of `Weight21` by the product of the learning rate, error of `Output2`, and the activation of `Hidden6`, we have finished updating the values of the weights for the network. We can now perform another forward pass using the new values of the weights; the value of the cost function produced using the updated weights should be smaller. We will repeat this process until the model converges or another stopping criterion is

satisfied. Unlike the linear models we have discussed, backpropagation does not optimize a convex function. It is possible that backpropagation will converge on parameter values that specify a local, rather than global, minimum. In practice, local optima are frequently adequate for many applications.

Approximating XOR with Multilayer perceptrons

Let's train a multilayer perceptron to approximate the XOR function. At the time of writing, multilayer perceptrons have been implemented as part of a 2014 Google Summer of Code project, but have not been merged or released. Subsequent versions of scikit-learn are likely to include this implementation of multilayer perceptrons without any changes to the API described in this section. In the interim, a fork of scikit-learn 0.15.1 that includes the multilayer perceptron implementation can be cloned from

<https://github.com/IssamLaradji/scikit-learn.git>.

First, we will create a toy binary classification dataset that represents XOR and split it into training and testing sets:

```
>>> from sklearn.cross_validation import  
train_test_split  
>>> from sklearn.neural_network import  
MultilayerPerceptronClassifier  
>>> y = [0, 1, 1, 0] * 1000  
>>> X = [[0, 0], [0, 1], [1, 0], [1, 1]] * 1000  
>>> X_train, X_test, y_train, y_test =  
train_test_split(X, y, random_state=3)
```

Next we instantiate `MultilayerPerceptronClassifier`. We specify the architecture of the network through the `n_hidden` keyword argument, which takes a list of the number of hidden units in each hidden layer. We create a hidden layer with two units that use the logistic activation function. The `MultilayerPerceptronClassifier` class automatically creates two input units and one output unit. In multi-class problems the classifier will create one output unit for each of the possible classes.

Selecting an architecture is challenging. There are some rules of thumb to choose the numbers of hidden units and layers, but these tend to be supported only by anecdotal evidence. The optimal number of hidden units depends on the number of training instances, the noise in the training data, the complexity of the function that is being approximated, the hidden units' activation function, the learning algorithm, and the regularization employed. In practice, architectures can only be evaluated by comparing their performances through cross validation.

We train the network by calling the `fit()` method:

```
>>> clf =  
MultilayerPerceptronClassifier(n_hidden=[2],  
>>>  
activation='logistic',  
>>>
```

```
algorithm='sgd',  
>>>  
random_state=3)  
>>> clf.fit(X_train, y_train)
```

Finally, we print some predictions for manual inspection and evaluate the model's accuracy on the test set. The network perfectly approximates the XOR function on the test set:

```
>>> print 'Number of layers: %s. Number of  
outputs: %s' % (clf.n_layers_, clf.n_outputs_)  
>>> predictions = clf.predict(X_test)  
>>> print 'Accuracy:', clf.score(X_test, y_test)  
>>> for i, p in enumerate(predictions[:10]):  
>>>     print 'True: %s, Predicted: %s' %  
(y_test[i], p)  
Number of layers: 3. Number of outputs: 1  
Accuracy: 1.0  
True: 1, Predicted: 1  
True: 1, Predicted: 1  
True: 1, Predicted: 1  
True: 0, Predicted: 0  
True: 1, Predicted: 1  
True: 0, Predicted: 0  
True: 0, Predicted: 0  
True: 1, Predicted: 1  
True: 0, Predicted: 0  
True: 1, Predicted: 1
```

Classifying handwritten digits

In the previous chapter we used a support vector machine to classify the handwritten digits in the MNIST dataset. In this section we will classify the images using an artificial neural network:

```
from sklearn.datasets import load_digits
from sklearn.cross_validation import
train_test_split, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from
sklearn.neural_network.mutilayer_perceptron
import MultilayerPerceptronClassifier
```

First we use the `load_digits` convenience function to load the MNIST dataset. We will fork additional processes during cross validation, which requires execution from a `main`-protected block:

```
>>> if __name__ == '__main__':
>>>     digits = load_digits()
>>>     X = digits.data
>>>     y = digits.target
```

Scaling the features is particularly important for artificial

neural networks and will help some learning algorithms to converge more quickly. Next, we create a `Pipeline` class that scales the data before fitting a

`MultilayerPerceptronClassifier`. This network contains an input layer, a hidden layer with 150 units, a hidden layer with 100 units, and an output layer. We also increased the value of the regularization hyperparameter `alpha` argument. Finally, we print the accuracies of the three cross validation folds. The code is as follows:

```
>>>     pipeline = Pipeline([
>>>         ('ss', StandardScaler()),
>>>         ('mlp',
MultilayerPerceptronClassifier(n_hidden=[150,
100], alpha=0.1))
>>>     ])
>>>     print cross_val_score(pipeline, X, y,
n_jobs=-1)
Accuracies [ 0.95681063  0.96494157  0.93791946]
```

The mean accuracy is comparable to the accuracy of the support vector classifier. Adding more hidden units or hidden layers and grid searching to tune the hyperparameters could further improve the accuracy.

Summary

In this chapter, we introduced artificial neural networks, powerful models for classification and regression that can represent complex functions by composing several artificial neurons. In particular, we discussed directed acyclic graphs of artificial neurons called feedforward neural networks. Multilayer perceptrons are a type of feedforward network in which each layer is fully connected to the subsequent layer. An MLP with one hidden layer and a finite number of hidden units is a universal function approximator. It can represent any continuous function, though it will not necessarily be able to learn appropriate weights automatically. We described how the hidden layers of a network represent latent variables and how their weights can be learned using the backpropagation algorithm. Finally, we used scikit-learn's multilayer perceptron implementation to approximate the function XOR and to classify handwritten digits.

This chapter concludes the book. We discussed a variety of models, learning algorithms, and performance measures, as well as their implementations in scikit-learn. In the first chapter, we described machine learning programs as those that learn from experience to improve their performance at a task. Then, we worked through

examples that demonstrated some of the most common experiences, tasks, and performance measures in machine learning. We regressed the prices of pizzas onto their diameters and classified spam and ham text messages. We clustered colors to compress images and clustered the SURF descriptors to recognize photographs of cats and dogs. We used principal component analysis for facial recognition, built a random forest to block banner advertisements, and used support vector machines and artificial neural networks for optical character recognition. Thank you for reading; I hope that you will be able to use scikit-learn and this book's examples to apply machine learning to your own experiences.

Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Learning scikit-learn: Machine Learning in Python*, Raúl Garreta and Guillermo Moncecchi
- *scikit-learn Cookbook*, Trent Hauck
- *Mastering Machine Learning with scikit-learn*, Gavin Hackeling

Index

A

- accuracy
 - about / [Performance measures, bias, and variance, Binary classification performance metrics, Accuracy](#)
- activation function
 - about / [Activation functions](#)
 - Heaviside step function / [Activation functions](#)
 - logistic sigmoid activation function / [Activation functions](#)
- Affinity Propagation
 - about / [Alternative clustering methods](#)
- Altman's Z-score / [How to do it...](#)
- artificial neural network / [Limitations of the perceptron](#)
- artificial neural networks
 - about / [Feedforward and feedback artificial neural networks](#)
- attributes, random forest
 - rf.criterion / [How to do it...](#)
 - rf.bootstrap / [How to do it...](#)
 - rf.n_jobs / [How to do it...](#)
 - rf.max_features / [How to do it...](#)

- rf.compute_idportances / [How to do it...](#)
 - rf.max_depth / [How to do it...](#)
- attributes, support vector classifier (SVC)
 - C / [How to do it...](#)
 - class_weight / [How to do it...](#)
 - gamma / [How to do it...](#)
 - kernel / [How to do it...](#)
- AUC
 - about / [ROC AUC](#)
- augmented term frequencies / [Extending bag-of-words with TF-IDF weights](#)
- automatic cross validation
 - about / [Getting ready, How it works...](#)

B

- backpropagation algorithm
 - about / [Minimizing the cost function, Backpropagation](#)
- bag-of-features
 - about / [Clustering to learn features](#)
- bag-of-words model
 - about / [The bag-of-words representation](#)
 - extending, with tf-idf weights / [Extending bag-of-words with TF-IDF weights](#)
- batch gradient descent / [Fitting models with gradient descent](#)
- Bayesian Ridge Regression
 - about / [Using Gaussian processes for regression](#)
- Bayesian ridge regression
 - applying, directly / [Directly applying Bayesian ridge regression, How it works..., There's more...](#)
- bell curve
 - about / [Binary classification with logistic regression](#)
- Bernoulli distribution
 - about / [Binary classification with logistic regression](#)
- bias / [Performance measures, bias, and variance](#)

- bias-variance trade-off
 - about / [Performance measures, bias, and variance](#)
- bias-variance tradeoff
 - about / [Important concepts related to machine learning](#)
- binary classification
 - with logistic regression / [Binary classification with logistic regression](#)
 - performance metrics / [Binary classification performance metrics](#)
 - accuracy / [Accuracy](#)
 - precision / [Precision and recall](#)
 - recall / [Precision and recall](#)
 - with perceptron / [Binary classification with the perceptron](#)
- binary features
 - creating, through thresholding / [Creating binary features through thresholding](#), [How it works...](#)
- bootstrapping
 - about / [There's more...](#) / [How it works...](#)
- bootstrap script
 - URL / [Installing scikit-learn on Windows](#)
- boston dataset
 - about / [Getting ready](#), [Getting ready](#)
- Brute force grid search

- about / [Brute force grid search](#)
- performing / [Getting ready](#), [How it works...](#)
- Bunch object
 - about / [How it works...](#)

C

- C4.5 / [Information gain](#)
- calc_params function / [Model selection, Grid search](#)
- categorical variables
 - about / [Working with categorical variables](#)
 - working with / [Getting ready, There's more...](#)
 - features, extracting from / [Extracting features from categorical variables](#)
- centroids
 - about / [Optimizing the number of centroids, Clustering with the K-Means algorithm](#)
 - optimizing / [Getting ready, How to do it..., How it works...](#)
- characters
 - classifying / [Classifying characters in scikit-learn](#)
 - handwritten digits, classifying / [Classifying handwritten digits](#)
 - in natural images, classifying / [Classifying characters in natural images](#)
- Chars74K
 - URL / [Classifying characters in natural images](#)
- classification
 - linear methods, using for / [Using linear methods for classification – logistic regression, How to](#)

[do it..., There's more...](#)

- about / [Introduction](#)
- LDA, using for / [How to do it..., How it works...](#)
- Stochastic Gradient Descent, using for / [Getting ready, How to do it...](#)
- classifications
 - performing, with Decision Trees / [Doing basic classifications with Decision Trees, How to do it..., How it works...](#)
- closest object
 - finding, in feature space / [Getting ready, How to do it..., How it works...](#)
- cluster
 - correctness, assessing / [Assessing cluster correctness, How to do it...](#)
- clustering / [Machine learning categories](#)
 - predicting, for training data / [Clustering handwritten digits with k-means](#)
 - method, alternatives / [Alternative clustering methods](#)
 - about / [Introduction, Using KMeans to cluster data](#)
 - with K-Means algorithm / [Clustering with the K-Means algorithm](#)
 - evaluation / [Evaluating clusters](#)
 - to learn features / [Clustering to learn features](#)

- clusters
 - about / [Clustering handwritten digits with k-means](#)
- cluster_centers_ attribute / [Alternative clustering methods](#)
- cluster_centers_indices_ attribute / [Alternative clustering methods](#)
- coef_ attribute / [Our first machine learning method – linear classification](#)
- Completeness
 - about / [Alternative clustering methods](#)
- compute_importances parameter / [How to do it...](#)
- confusion matrix
 - about / [Binary classification performance metrics](#)
- contingency table
 - about / [Binary classification performance metrics](#)
- convex hulls / [Limitations of the perceptron](#)
- corners / [Extracting points of interest as features](#)
- corpus
 - about / [The bag-of-words representation](#)
- correlation functions, scikit-learn / [How it works...](#)
- cosine kernel
 - about / [How to do it...](#)
- cost function
 - model fitness, evaluating / [Evaluating the](#)

fitness of a model with a cost function

- about / Evaluating the fitness of a model with a cost function
- minimizing / Minimizing the cost function
- CountVectorizer / Preprocessing the data
- CountVectorizer class / The bag-of-words representation, Stop-word filtering
- covariance
 - about / Solving ordinary least squares for simple linear regression, Variance, Covariance, and Covariance Matrices
- covariance matrix
 - about / Variance, Covariance, and Covariance Matrices
- covariance_type method / Alternative clustering methods
- cross-validation
 - about / Training data and test data
 - folds / Training data and test data
 - partitions / Training data and test data
- cross validation
 - about / How it works...
- cross validation, with ShuffleSplit / Getting ready, How to do it...
- cross_val_score helper function / Fitting and evaluating the model
- curse of dimensionality / The bag-of-words

representation

D

- data
 - preprocessing / [Preprocessing the data](#)
 - scaling, to standard normal / [Scaling data to the standard normal](#), [How it works...](#)
 - line, fitting through / [Fitting a line through data](#), [How to do it...](#), [How it works...](#)
 - clustering, KMeans used / [Getting ready](#), [How to do it...](#), [How it works...](#)
 - handling, MiniBatch KMeans used / [How to do it...](#), [How it works...](#)
 - classifying, with Support Vector Machines (SVM) / [Getting ready](#), [How it works...](#)
 - exploring / [Exploring the data](#)
- data array / [Datasets](#)
- DataFrame / [Feature extraction](#)
- Dataframe.describe() method / [Exploring the data](#)
- data imputation
 - about / [Imputing missing values through various strategies](#)
- data preprocessing
 - about / [Important concepts related to machine learning](#)
- data set
 - URL / [Applying linear regression](#)

- dataset
 - URL / [Image recognition with Support Vector Machines](#)
- datasets
 - about / [Datasets](#)
- datasets module
 - about / [How to do it...](#)
- data standardization
 - about / [Data standardization](#)
- decision boundary / [How it works...](#)
- decision tree, versus random forest
 - size, checking / [There's more...](#)
- Decision Tree model
 - tuning / [Tuning a Decision Tree model, How to do it...](#)
- decision trees
 - Titanic hypothesis with / [Explaining Titanic hypothesis with decision trees](#)
 - about / [Explaining Titanic hypothesis with decision trees, Decision trees](#)
 - data, preprocessing with / [Preprocessing the data](#)
 - classifier, training / [Training a decision tree classifier](#)
 - interpreting / [Interpreting the decision tree](#)
 - performance, evaluating / [Evaluating the performance](#)

- training / [Training decision trees](#)
- questions, selecting / [Selecting the questions](#)
- information gain / [Information gain](#)
- Gini impurity / [Gini impurity](#)
- with scikit-learn / [Decision trees with scikit-learn](#)
- tree ensembles / [Tree ensembles](#)
- advantages / [The advantages and disadvantages of decision trees](#)
- disadvantages / [The advantages and disadvantages of decision trees](#)
- eager learners / [The advantages and disadvantages of decision trees](#)
- lazy learners / [The advantages and disadvantages of decision trees](#)
- Decision Trees
 - used, for performing classifications / [Doing basic classifications with Decision Trees, How to do it..., How it works...](#)
- decomposition
 - factor analysis, using for / [Getting ready, How to do it...](#)
- dictionary
 - about / [The bag-of-words representation](#)
- DictionaryLearning
 - about / [Decomposition to classify with DictionaryLearning](#)

- decomposition, performing for classification / [Getting ready](#), [How to do it...](#), [How it works...](#)
- DictVectorizer class / [Extracting features from categorical variables](#)
- DictVectorizer option / [DictVectorizer](#)
- digit class / [Principal Component Analysis](#)
- dimensionality
 - reducing, with PCA / [Getting ready](#), [How it works...](#)
 - reducing, truncated SVD used / [Getting ready](#), [How to do it...](#), [How it works...](#)
- dimensionality reduction
 - about / [Machine learning tasks](#)
- dimensionality reduction
 - with PCA / [Dimensionality reduction with Principal Component Analysis](#)
- distance functions
 - about / [How it works...](#)
- document classification
 - with perceptron / [Document classification with the perceptron](#)
- documents
 - classifying, with Naïve Bayes / [Getting ready](#), [How to do it...](#)
- dual form
 - about / [Kernels and the kernel trick](#)
- dummy estimators

- used, for comparing results / [Using dummy estimators to compare results](#), [How to do it...](#), [How it works...](#)
- dunder / [There's more...](#)

E

- eager learners / [The advantages and disadvantages of decision trees](#)
- edges / [Extracting points of interest as features](#)
- effective rank
 - about / [How to do it...](#)
- eigenfaces / [Face recognition with PCA](#)
- eigenvalue
 - about / [Eigenvectors and eigenvalues](#)
- eigenvector
 - about / [Eigenvectors and eigenvalues](#)
- elastic net regularization
 - about / [Regularization](#)
- elbow method, K-Means algorithm
 - about / [The elbow method](#)
- embarked attribute / [Feature extraction](#)
- embarrassingly parallel problem / [Tuning models with grid search](#)
- ensemble learning / [Tree ensembles](#)
- entropy
 - versus Gini impurity / [How it works...](#)
 - about / [Selecting the questions](#)
- epoch
 - about / [The perceptron learning algorithm](#)
- error-driven learning algorithm

- about / [The perceptron learning algorithm](#)
- estimators
 - about / [Simple linear regression](#)
- Euclidean distance / [The bag-of-words representation](#)
- Euclidean norm / [The bag-of-words representation](#)
- Expectation-Maximization (EM) / [Alternative clustering methods](#)
- explanatory variables / [Learning from experience](#)
- external sources
 - sample data, obtaining from / [Getting sample data from external sources, How to do it..., There's more...](#)
- ExtraTreesRegressor class / [Third try – Random Forests revisited](#)

F

- =F1-score / [Evaluating our results](#)
- F1 measure
 - about / [Binary classification performance metrics](#)
 - calculating / [Calculating the F1 measure](#)
- face recognition
 - with PCA / [Face recognition with PCA](#)
- faces object / [Image recognition with Support Vector Machines](#)
- factor analysis
 - about / [Using factor analysis for decomposition](#)
 - using, for decomposition / [Getting ready, How to do it...](#)
- fall-out
 - about / [ROC AUC](#)
- false positive
 - about / [Performance measures, bias, and variance](#)
- feature engineering
 - about / [Important concepts related to machine learning](#)
- feature extraction
 - about / [Feature extraction](#)
 - obtain features / [Feature extraction](#)

- convert features / [Feature extraction](#)
- feature importance / [There's more...](#)
- features
 - extracting, from categorical variables / [Extracting features from categorical variables](#)
 - extracting, from text / [Extracting features from text](#)
 - extracting, from images / [Extracting features from images](#)
 - extracting, from pixel intensities / [Extracting features from pixel intensities](#)
 - points of interest, extracting as / [Extracting points of interest as features](#)
- feature selection
 - about / [Feature selection](#), [Feature selection](#), [How to do it...](#), [How it works...](#)
- feature selection, on L1 norms / [Feature selection on L1 norms](#), [How to do it...](#), [How it works...](#)
- feature space
 - closest objects, finding in / [Getting ready](#), [How to do it...](#), [How it works...](#)
- feature_selection module / [Feature selection](#)
 - importing / [How to do it...](#)
- feedback artificial neural networks
 - about / [Feedforward and feedback artificial neural networks](#)
- feedback neural networks

- about / [Feedforward and feedback artificial neural networks](#)
- Feedforward neural networks
 - about / [Feedforward and feedback artificial neural networks](#)
- fit method / [Clustering handwritten digits with k-means](#)
 - about / [The fit method](#)
- forward propagation
 - about / [Forward propagation](#)
- functional margin / [Maximum margin classification and support vectors](#)

G

- Gauss-Markov theorem
 - about / [How it works...](#)
- Gaussian distribution
 - about / [Binary classification with logistic regression](#)
- Gaussian kernel
 - about / [Kernels and the kernel trick](#)
- Gaussian Mixture Models
 - probabilistic clustering, performing with /
[Getting ready](#), [How to do it...](#), [How it works...](#)
- Gaussian Mixture Models (GMM)
 - about / [Alternative clustering methods](#)
- Gaussian process
 - about / [Using Gaussian processes for regression](#)
 - using, for regression / [Using Gaussian processes for regression](#), [How to do it...](#), [How it works...](#)
- GaussianProcess object
 - beta0 / [How to do it...](#)
 - corr / [How to do it...](#)
 - regr / [How to do it...](#)
 - nugget / [How to do it...](#)
 - normalize / [How to do it...](#)
- Gaussian process object
 - defining / [How to do it...](#)

- gaussian_process module
 - about / [Getting ready](#)
- geometric margin
 - about / [Maximum margin classification and support vectors](#)
- Gini impurity
 - versus entropy / [How it works...](#)
 - about / [How it works...](#)
 - / [Gini impurity](#)
- gradient boosting regression
 - about / [Getting ready](#)
 - working / [How to do it...](#), [How it works...](#)
- gradient descent
 - models, fitting with / [Fitting models with gradient descent](#)
- Graphviz
 - URL / [Training a decision tree classifier](#)
- greedy / [The advantages and disadvantages of decision trees](#)
- grid search
 - about / [Grid search](#)
 - parallel grid search / [Parallel grid search](#)
 - performing / [Getting ready](#), [How to do it...](#)
 - models, tuning with / [Tuning models with grid search](#)

H

- Hamming loss
 - about / [Multi-label classification performance metrics](#)
- handwritten digits
 - classifying / [Classifying handwritten digits](#),
[Classifying handwritten digits](#)
- harmonic mean / [Evaluating our results](#)
- hashing trick
 - space-efficient feature, vectorizing / [Space-efficient feature vectorizing with the hashing trick](#)
- HashingVectorizer / [Preprocessing the data](#)
- Heaviside step function
 - about / [Activation functions](#)
- hidden layer, MLP / [Multilayer perceptrons](#)
- high-dimensional data
 - visualizing, PCA used / [Using PCA to visualize high-dimensional data](#)
- hold-out set
 - about / [Training data and test data](#)
- Homogeneity
 - about / [Alternative clustering methods](#)
- house prices
 - predicting, with regression / [Predicting house](#)

prices with regression

- linear model / [First try – a linear model](#)
- Support Vector Machines / [Second try – Support Vector Machines for regression](#)
- Random Forests / [Third try – Random Forests revisited](#)
 - performance, evaluating / [Evaluation](#)
- Hughes effect / [The bag-of-words representation](#)
- hyperparameters
 - about / [Training data and test data](#)
- Hyperparameters
 - about / [Regularization](#)

I

- idempotent scalar objects
 - creating / [Creating idempotent scalar objects](#)
- identity link function
 - about / [Binary classification with logistic regression](#)
- image
 - quantizing, with KMeans clustering / [Getting ready, How do it...](#)
 - quantization / [Image quantization](#)
- image recognition
 - with Support Vector Machines / [Image recognition with Support Vector Machines](#)
- imputation, scikit-learn
 - idempotent scalar objects. creating / [Creating idempotent scalar objects](#)
 - sparse imputations, handling / [Handling sparse imputations](#)
- inertia
 - about / [There's more...](#)
- information gain
 - about / [Information gain](#)
- Information Gain (IG) / [Training a decision tree classifier, How it works...](#)
- input layer, MLP / [Multilayer perceptrons](#)

- intercept_ attribute / [Our first machine learning method – linear classification](#)
- Internet Advertisements Data Set
 - URL / [Decision trees with scikit-learn](#)
- inverse document frequency (IDF) / [Extending bag-of-words with TF-IDF weights](#)
- IPython Notebook / [Installing scikit-learn](#)
- IPython parallel / [Parallel grid search](#)
- Iterative Dichotomiser 3 (ID3)
 - about / [Training decision trees](#)
- IterGrid iterator / [Parallel grid search](#)

J

- Jaccard similarity
 - about / [Multi-label classification performance metrics](#)
- joblib
 - models, persisting with / [Persisting models with joblib, How it works...](#)

K

- k-fold cross validation
 - about / [K-fold cross validation, How it works...](#)
- k-means
 - about / [Clustering handwritten digits with k-means](#)
- K-Means algorithm
 - clustering with / [Clustering with the K-Means algorithm](#)
 - local optima / [Local optima](#)
 - elbow method / [The elbow method](#)
- k-NN
 - using, for regression / [Getting ready, How to do it...](#)
- kernelization / [Limitations of the perceptron](#)
- kernel keyword argument / [Classifying handwritten digits](#)
- kernel PCA, nonlinear dimensionality reduction / [Kernel PCA for nonlinear dimensionality reduction, How to do it..., How it works...](#)
- kernels
 - about / [Kernels and the kernel trick](#)
 - polynomial kernels / [Kernels and the kernel trick](#)
 - Quadratic kernels / [Kernels and the kernel trick](#)

- sigmoid kernel / [Kernels and the kernel trick](#)
 - Gaussian kernel / [Kernels and the kernel trick](#)
- kernel trick
 - about / [Kernels and the kernel trick](#)
- KMeans
 - about / [Using KMeans to cluster data](#), [Using MiniBatch KMeans to handle more data](#)
 - used, for clustering data / [Getting ready](#), [How to do it...](#), [How it works...](#)
 - using, for outlier detection / [Getting ready](#), [How to do it...](#), [How it works...](#)
- KMeans clustering
 - image, quantizing with / [Getting ready](#), [How do it...](#)

L

- LabelBinarizer() method / [How to do it...](#)
- label features
 - binarizing / [Binarizing label features, How it works...](#), [There's more...](#)
- label propagation
 - about / [Label propagation with semi-supervised learning](#)
- label propagation, semi-supervised learning / [Getting ready](#), [How to do it...](#)
- labels_ attribute / [Clustering handwritten digits with k-means](#)
- LARS
 - about / [Getting ready](#)
- LASSO
 - about / [Regularization](#)
- Lasso, feature selection
 - about / [Lasso for feature selection](#)
- Lasso cross-validation
 - about / [Lasso cross-validation](#)
- Law of Large Numbers
 - about / [Alternative clustering methods](#)
- lazy learners / [The advantages and disadvantages of decision trees](#)
- LDA

- using, for classification / [How to do it...](#), [How it works...](#)
- least absolute shrinkage and selection operator (LASSO)
 - about / [Using sparsity to regularize models](#)
- leave-one-out cross-validation / [Interpreting the decision tree](#)
- leave-one-out cross-validation (LOOCV)
 - about / [How to do it...](#)
- lemma / [Stemming and lemmatization](#)
- lemmatization / [Stemming and lemmatization](#)
- line
 - fitting, through data / [Fitting a line through data](#), [How to do it...](#), [How it works...](#)
- linear classification
 - about / [Our first machine learning method – linear classification](#)
- linear least squares / [Simple linear regression](#)
- linearly separable datasets / [Limitations of the perceptron](#)
- linear methods
 - using, for classification / [Using linear methods for classification – logistic regression](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- linear models
 - about / [Introduction](#)
- linear regression

- about / [Fitting a line through data](#)
 - simple linear regression / [Simple linear regression](#)
 - multiple linear regression / [Multiple linear regression](#)
 - applying / [Applying linear regression](#)
 - data, exploring / [Exploring the data](#)
- linear regression model
 - evaluating / [Evaluating the linear regression model](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- LinearRegression object
 - about / [There's more...](#)
- linear_models module / [How to do it...](#)
- Linux
 - Scikit-learn, installing on / [Verifying the installation](#)
 - scikit-learn, installing / [Installing scikit-learn on Linux](#)
- local optima, K-Means algorithm
 - about / [Local optima](#)
- logarithmically scaled term frequencies / [Extending bag-of-words with TF-IDF weights](#)
- logistic function
 - about / [Binary classification with logistic regression](#)
- logistic regression

- about / [Using linear methods for classification – logistic regression](#)
- binary classification with / [Binary classification with logistic regression](#)
- LogisticRegression classifier / [How to do it...](#)
- LogisticRegression object
 - about / [How to do it...](#)
- logistic sigmoid activation function
 - about / [Activation functions](#)
- logit function
 - about / [Binary classification with logistic regression](#)
- loss function / [How it works...](#)
 - about / [Evaluating the fitness of a model with a cost function](#)
- lr object
 - about / [Getting ready](#)
- ls parameter / [How it works...](#)

M

- %matplotlib inline command / [Getting sample data from external sources](#)
- Mac
 - Scikit-learn, installing on / [Mac](#)
- machine learning
 - method / [Our first machine learning method – linear classification](#)
 - linear classification / [Our first machine learning method – linear classification](#)
 - results, evaluating / [Evaluating our results](#)
 - concepts / [Important concepts related to machine learning](#)
 - tasks / [Machine learning tasks](#)
 - regression task / [Machine learning tasks](#)
- machine learning (ML) / [Introduction](#)
- machine learning categories
 - about / [Machine learning categories](#)
- margin
 - classification / [Maximum margin classification and support vectors](#)
- matplotlib
 - URL / [Installing pandas and matplotlib](#)
- matplotlib package
 - URL, for installing / [Installing scikit-learn](#)

- max_depth parameter / [How it works...](#), [How to do it...](#)
- mean absolute deviation (MAD)
 - about / [How to do it...](#), [How it works...](#)
- mean squared error (MSE)
 - about / [How to do it...](#), [How it works...](#)
- measure_performance function / [Evaluation](#)
- meshgrid
 - of points / [Clustering handwritten digits with k-means](#)
- MiniBatch KMeans
 - used, for handling data / [How to do it...](#), [How it works...](#)
- missing values
 - imputing, through various strategies / [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- MLP
 - about / [Multilayer perceptrons](#)
 - input layer / [Multilayer perceptrons](#)
 - hidden layer / [Multilayer perceptrons](#)
 - output layer / [Multilayer perceptrons](#)
 - XOR, approximating with / [Approximating XOR with Multilayer perceptrons](#)
- model
 - fitting / [Fitting and evaluating the model](#)
 - evaluating / [Fitting and evaluating the model](#),

Evaluating the model

- fitness, evaluating with cost function /
Evaluating the fitness of a model with a cost function
- models
 - regularizing, sparsity used / Using sparsity to regularize models, How it works...
 - persisting, with joblib / Persisting models with joblib, How it works...
 - fitting, with gradient descent / Fitting models with gradient descent
 - tuning, with grid search / Tuning models with grid search
- model selection
 - about / Interpreting the decision tree, Model selection
- multi-class classification
 - about / Multi-class classification
 - performance metrics / Multi-class classification performance metrics
- multi-label classification
 - and problem transformation / Multi-label classification and problem transformation
 - performance metrics / Multi-label classification performance metrics
- multiclass classification
 - generalizing with / Getting ready, How it

works...

- MultinomialNB algorithm / [Model selection](#)
- multiple linear regression
 - about / [Multiple linear regression](#)
- multiple preprocessing steps
 - Pipelines, using for / [Getting ready](#), [How to do it...](#), [How it works...](#)

N

- n-init parameter / [Clustering handwritten digits with k-means](#)
- natural images
 - characters, classifying / [Classifying characters in natural images](#)
- Natural Language Tool Kit (NLTK)
 - URL / [Stemming and lemmatization](#)
- Naïve Bayes
 - about / [Text classification with Naïve Bayes, Classifying documents with Naïve Bayes](#)
 - used, for classifying text / [Text classification with Naïve Bayes](#)
 - data, preprocessing / [Preprocessing the data](#)
 - classifier, training / [Training a Naïve Bayes classifier](#)
 - performance, evaluating / [Evaluating the performance](#)
 - documents, classifying with / [Getting ready, How to do it...](#)
 - extending / [There's more...](#)
- NLP
 - about / [Text classification with Naïve Bayes](#)
- nonlinear decision boundaries
 - about / [Nonlinear decision boundaries](#)

- normal distribution
 - about / [Binary classification with logistic regression](#)
- normalization
 - about / [How it works...](#)
- NP-hard
 - about / [Getting ready](#)
- NumPy
 - 32-bit version, URL / [Installing scikit-learn on Windows](#)
 - 64-bit version, URL / [Installing scikit-learn on Windows](#)

O

- one-vs.-all
 - about / [Multi-class classification](#)
- one-vs.-the-rest
 - about / [Multi-class classification](#)
- OneHotEncoder class / [Preprocessing the data](#)
- one hot encoding / [Preprocessing the data](#)
- OneVsRestClassifier / [Getting ready](#)
- one_hot_dataframe method / [Feature extraction](#)
- Optical character recognition (OCR)
 - about / [Extracting features from pixel intensities](#)
- ordinary least squares / [Simple linear regression](#)
 - solving, for simple linear regression / [Solving ordinary least squares for simple linear regression](#)
- OS X
 - scikit-learn, installing / [Installing scikit-learn on OS X](#)
- outlier detection
 - KMeans, using for / [Getting ready, How to do it..., How it works...](#)
- output layer, MLP / [Multilayer perceptrons](#)
- over-fitting
 - about / [Training data and test data, Polynomial regression](#)

- overfitting / Evaluating our results

P

- pairwise distances
 - about / [There's more...](#)
- pairwise_distances function
 - about / [How to do it...](#)
- pandas
 - installing / [Installing pandas and matplotlib](#)
- parallel_grid_search function / [Parallel grid search](#)
- partial_fit() method / [Document classification with the perceptron](#)
- patsy option / [Patsy](#)
- PCA
 - about / [Principal Component Analysis](#),
[Reducing dimensionality with PCA](#), [An overview of PCA](#)
 - visualization / [Principal Component Analysis](#)
 - feature, selecting / [Principal Component Analysis](#)
 - function, defining / [Principal Component Analysis](#)
 - dimensionality, reducing with / [Getting ready](#),
[How it works...](#)
 - performing / [Performing Principal Component Analysis](#)
 - using, to visualize high-dimensional data /

Using PCA to visualize high-dimensional data

- face recognition with / Face recognition with PCA
- PCA, performing
 - variance / Variance, Covariance, and Covariance Matrices
 - covariance / Variance, Covariance, and Covariance Matrices
 - covariance matrix / Variance, Covariance, and Covariance Matrices
 - eigenvector / Eigenvectors and eigenvalues
 - eigenvalue / Eigenvectors and eigenvalues
 - dimensionality reduction / Dimensionality reduction with Principal Component Analysis
- PCA object
 - about / There's more...
- pclass attribute / Feature extraction
- pd.read_csv() function / Exploring the data
- perceptron
 - error-driven learning algorithm / The perceptron learning algorithm
 - learning algorithm / The perceptron learning algorithm
 - binary classification with / Binary classification with the perceptron
 - document classification with / Document classification with the perceptron

- limitations / [Limitations of the perceptron](#)
- perceptron's preactivation / [Activation functions](#)
- perceptron learning algorithm
 - about / [The perceptron learning algorithm](#)
- performance
 - measures / [Performance measures, bias, and variance](#)
- performance metrics
 - binary classification / [Binary classification performance metrics](#)
 - multi-class classification / [Multi-class classification performance metrics](#)
- Pipeline class / [Evaluating our results](#)
- Pipelines
 - about / [Using Pipelines for multiple preprocessing steps](#)
 - using, for multiple preprocessing steps / [Getting ready, How to do it..., How it works...](#)
 - working / [Getting ready, How to do it..., How it works..., There's more...](#)
- pixel intensities
 - features, extracting from / [Extracting features from pixel intensities](#)
- points of interest
 - extracting, as features / [Extracting points of interest as features](#)
 - edges / [Extracting points of interest as features](#)

- corners / [Extracting points of interest as features](#)
- polynomial kernels
 - about / [Kernels and the kernel trick](#)
- polynomial regression
 - about / [Polynomial regression](#)
- preactivation / [Forward propagation](#)
- precision / [Evaluating our results](#)
 - about / [Performance measures, bias, and variance, Binary classification performance metrics, Precision and recall](#)
- precision parameter / [How to do it...](#)
- prediction errors
 - about / [Evaluating the fitness of a model with a cost function](#)
- preprocessing module
 - about / [Getting ready](#)
- primal form
 - about / [Kernels and the kernel trick](#)
- principal components
 - about / [An overview of PCA](#)
- print_digits function / [Clustering handwritten digits with k-means, Alternative clustering methods](#)
- print_faces function / [Fitting and evaluating the model](#)
- probabilistic clustering
 - performing, with Gaussian Mixture Models / [Getting ready, How to do it..., How it works...](#)

- problem transformation
 - and multi-label classification / [Multi-label classification and problem transformation](#)
- pruning / [Information gain](#), [The advantages and disadvantages of decision trees](#)
- pydot / [Getting ready](#)
- pydot module / [Training a decision tree classifier](#)
- Python
 - URL / [Installing scikit-learn](#)
- Python package pandas
 - URL / [Feature extraction](#)

Q

- QDA
 - about / [Working with QDA – a nonlinear LDA](#)
 - working with / [How to do it..., How it works...](#)
- Quadratic kernels
 - about / [Kernels and the kernel trick](#)
- questioners / [Decision trees](#)
- questions, decision trees
 - selecting / [Selecting the questions](#)

R

- r-squared measures
 - about / [Evaluating the model](#)
- radial basis function
 - using / [There's more...](#)
 - about / [Kernels and the kernel trick](#)
- Rand index
 - about / [Clustering handwritten digits with k-means](#)
- random forest / [Tree ensembles](#)
- random forest model
 - tuning / [Tuning a random forest model](#), [How to do it...](#)
- Random Forests
 - about / [Random Forests – randomizing decisions](#)
- random forests
 - using / [Using many Decision Trees – random forests](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
- recall / [Evaluating our results](#)
 - about / [Performance measures, bias, and variance](#), [Binary classification performance metrics](#), [Precision and recall](#)
- recall parameter / [How to do it...](#)

- regression
 - Gaussian process, using for / [Using Gaussian processes for regression](#), [How to do it...](#), [How it works...](#)
 - Stochastic Gradient Descent (SGD), using for / [Getting ready](#), [How to do it...](#), [How it works...](#)
 - about / [Using k-NN for regression](#)
 - k-NN, using for / [Getting ready](#), [How to do it...](#)
- regression model
 - evaluating / [Regression model evaluation](#), [How to do it...](#), [How it works...](#)
- regression task, machine learning / [Machine learning tasks](#)
- regularization
 - about / [Regularization](#)
- regularization, LARS
 - about / [How to do it...](#), [How it works...](#)
- replace parameter / [Feature extraction](#)
- residuals
 - about / [How to do it...](#), [Evaluating the fitness of a model with a cost function](#)
- residual sum of squares
 - about / [Evaluating the fitness of a model with a cost function](#)
- response variable / [Learning from experience](#)
- results
 - comparing, dummy estimators used / [Using](#)

[dummy estimators to compare results](#), [How to do it...](#), [How it works...](#)

- ridge cross-validation
 - about / [How to do it...](#)
- RidgeCV object / [How to do it...](#)
- Ridge regression / [Regularization](#)
- ridge regression
 - used, for overcoming linear regression's shortfalls / [Using ridge regression to overcome linear regression's shortfalls](#), [How to do it...](#), [How it works...](#)
- ridge regression parameter
 - optimizing / [Optimizing the ridge regression parameter](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- ROC AUC
 - about / [Binary classification performance metrics](#), [ROC AUC](#)
- ROC curve
 - about / [ROC AUC](#)
- root-mean-square deviation (RMSE)
 - about / [How it works...](#)

S

- sample data
 - obtaining, from external sources / [Getting sample data from external sources](#), [How to do it...](#), [There's more...](#)
 - creating, for toy analysis / [Getting ready](#), [How to do it...](#), [How it works...](#)
- scikit-image
 - about / [Quantizing an image with KMeans clustering](#)
- Scikit-learn
 - installing / [Installing scikit-learn](#)
 - installing, on Linux / [Verifying the installation](#)
 - installing, on Mac / [Mac](#)
 - installing, on Windows / [Windows](#)
 - installation, checking / [Checking your installation](#)
- scikit-learn
 - installation, verifying / [Verifying the installation](#)
 - URL / [How it works...](#)
 - about / [An introduction to scikit-learn](#)
 - installing / [Installing scikit-learn](#)
 - installing, on Windows / [Installing scikit-learn on Windows](#)

- installing, on Linux / [Installing scikit-learn on Linux](#)
- installing, on OS X / [Installing scikit-learn on OS X](#)
- decision trees with / [Decision trees with scikit-learn](#)
- characters, classifying / [Classifying characters in scikit-learn](#)
- scikit-learn tutorial
 - URL / [Clustering handwritten digits with k-means](#)
- SelectPercentile method / [Feature selection](#)
- semi-supervised learning
 - problems / [Learning from experience](#)
- semi-supervised technique / [Label propagation with semi-supervised learning](#)
- Sequential Minimal Optimization (SMO)
 - about / [Maximum margin classification and support vectors](#)
- SGD / [Our first machine learning method –linear classification](#)
- SGDClassifier initialization function / [Our first machine learning method –linear classification](#)
- ShuffleSplit
 - about / [Cross validation with ShuffleSplit](#)
 - used, for performing cross validation / [Getting ready, How to do it...](#)

- SIFT
 - about / [SIFT and SURF](#)
- sigmoid kernel / [Kernels and the kernel trick](#)
- silhouette coefficient
 - about / [Evaluating clusters](#)
- silhouette distance
 - about / [How to do it...](#)
- simple linear regression
 - about / [Simple linear regression](#)
 - ordinary least squares, solving for / [Solving ordinary least squares for simple linear regression](#)
- sklearn.datasets module / [Text classification with Naïve Bayes](#)
- sklearn.decomposition module / [Principal Component Analysis](#)
- sklearn.ensemble module / [Random Forests – randomizing decisions, Third try – Random Forests revisited](#)
- sklearn.feature_extraction.text module / [Preprocessing the data](#)
- sklearn.grid_search module / [Grid search](#)
- sklearn.metrics.pairwise
 - about / [Getting ready](#)
- sklearn.naive_bayes module / [Training a Naïve Bayes classifier](#)
- sklearn.pipeline module / [Training a Naïve Bayes](#)

[classifier](#)

- sklearn.svm module / [Fitting and evaluating the model](#)
- sklearn package
 - about / [How to do it...](#)
- SMS Spam Classification Data Set
 - URL / [Spam filtering](#)
- space-efficient feature
 - vectorizing, with hashing trick / [Space-efficient feature vectorizing with the hashing trick](#)
- spam filtering
 - about / [Text classification with Naïve Bayes, Spam filtering](#)
- sparse imputations
 - handling / [Handling sparse imputations](#)
- sparse matrices
 - about / [Sparse matrices](#)
- sparse vectors / [The bag-of-words representation](#)
- sparsity
 - used, for regularizing models / [Getting ready, How it works...](#)
- spherical clusters
 - about / [Getting ready](#)
- standard normal
 - about / [Scaling data to the standard normal](#)
 - data, scaling to / [Scaling data to the standard normal, How it works...](#)

- stemming / [Stemming and lemmatization](#)
- Stochastic Gradient Descent
 - using, for classification / [Getting ready](#), [How to do it...](#)
- Stochastic Gradient Descent (SGD)
 - using, for regression / [Getting ready](#), [How to do it...](#), [How it works...](#)
- / [Fitting models with gradient descent](#)
- Stop-word filtering
 - about / [Stop-word filtering](#)
- stop words / [Stop-word filtering](#)
- strategies
 - missing values, imputing through / [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- stratified k-fold valuation
 - viewing / [Stratified k-fold](#), [How to do it...](#), [How it works...](#)
- supervised learning program / [Learning from experience](#)
- support vector classifier (SVC)
 - about / [How to do it...](#)
- Support Vector Machines (SVM)
 - about / [Classifying data with support vector machines](#)
 - data, classifying with / [Getting ready](#), [How it works...](#)

- support vectors
 - about / [Classifying data with support vector machines](#), [Maximum margin classification and support vectors](#)
- SURF
 - about / [SIFT and SURF](#)
- survived feature / [Feature extraction](#)
- SVC
 - image recognition with / [Image recognition with Support Vector Machines](#)
 - training / [Fitting and evaluating the model](#)
 - about / [Fitting and evaluating the model](#)
 - data, reshaping / [Fitting and evaluating the model](#)

T

- target array / [Datasets](#)
- test errors
 - about / [Evaluating the fitness of a model with a cost function](#)
- test set / [Learning from experience](#)
- text
 - features, extracting from / [Extracting features from text](#)
- TF-IDF
 - about / [Preprocessing the data](#)
- tf-idf weights
 - bag-of-words model, extending / [Extending bag-of-words with TF-IDF weights](#)
- TfidfVectorizer / [Training a Naïve Bayes classifier](#)
- thresholding
 - binary features, creating through / [Creating binary features through thresholding, How it works...](#)
- Tikhonov regularization / [Regularization](#)
- Titanic dataset / [Explaining Titanic hypothesis with decision trees](#)
- tokens
 - about / [The bag-of-words representation](#)
- toy analysis

- sample data, creating for / [Getting ready](#), [How to do it...](#), [How it works...](#)
- training errors
 - about / [Evaluating the fitness of a model with a cost function](#)
- training set / [Learning from experience](#)
- train_test_split function / [Our first machine learning method –linear classification](#)
- TruncatedSVD
 - about / [There's more...](#)
 - sign flipping / [Sign flipping](#)
 - sparse matrices / [Sparse matrices](#)
- true positive
 - about / [Performance measures, bias, and variance](#)
- truncated SVD
 - used, for reducing dimensionality / [Getting ready](#), [How to do it...](#), [How it works...](#)

U

- UCI Machine Learning Repository / [See also](#)
- units / [Multilayer perceptrons](#)
- unit variance
 - about / [Data standardization](#)
- univariate selection
 - about / [Feature selection](#)
- universal function approximators / [Limitations of the perceptron](#)

V

- validation
 - about / [Training data and test data](#)
- variance / [Performance measures, bias, and variance](#)
 - about / [Solving ordinary least squares for simple linear regression, Variance, Covariance, and Covariance Matrices](#)
- VarianceThreshold object / [How to do it...](#)
- vector's dimension / [The bag-of-words representation](#)

W

- Windows
 - Scikit-learn, installing on / [Windows](#)
 - scikit-learn, installing / [Installing scikit-learn on Windows](#)
- Windows installer
 - 32-bit version of scikit-learn, URL / [Installing scikit-learn on Windows](#)
 - 64-bit version of scikit-learn, URL / [Installing scikit-learn on Windows](#)

X

- XOR / [Limitations of the perceptron](#)
 - approximating, with MLP / [Approximating XOR with Multilayer perceptrons](#)

Z

- z-scores
 - about / [Scaling data to the standard normal](#)
- zero mean
 - about / [Data standardization](#)