Python: Deeper Insights into Machine Learning

Leverage benefits of machine learning techniques using Python

Python: Deeper Insights into Machine Learning

Leverage benefits of machine learning techniques using Python

A course in three modules

BIRMINGHAM - MUMBAI

Python: Deeper Insights into Machine Learning

Credits

Authors

   Sebastian Raschka

   David Julian

   John Hearty

Reviewers

   Richard Dutton

   Dave Julian

   Vahid Mirjalili

   Hamidreza Sattari

   Dmytro Taranovsky

   Dr. Vahid Mirjalili

   Jared Huffman

   Ashwin Pajankar

Content Development Editor

   Amrita Noronha

Production Coordinator

   Arvindkumar Gupta

Preface

Machine learning and predictive analytics are becoming one of the key strategies for unlocking growth in a challenging contemporary marketplace .It is one of the fastest growing trends in modern computing and everyone wants to get into the field of machine learning. In order to obtain sufficient recognition in this field, one must be able to understand and design a machine learning system that serves the needs of a project. The idea is to prepare a Learning Path that will help you to tackle the real-world complexities of modern machine learning with innovative and cutting-edge techniques. Also, it will give you a solid foundation in the machine learning design process, and enable you to build customized machine learning models to solve unique problems

What this learning path covers

Module 1, Python Machine Learning, discusses the essential machine algorithms for classification and provides practical examples using scikit-learn. It teaches you to prepare variables of different types and also speaks about polynomial regression and tree-based approaches. This module focuses on open source Python library that allows us to utilize multiple cores of modern GPUs.

Module 2, Designing Machine Learning Systems with Python, acquaints you with large library of packages for machine learning tasks. It introduces broad topics such as big data, data properties, data sources, and data processing .You will further explore models that form the foundation of many advanced nonlinear techniques. This module will help you in understanding model selection and parameter tuning techniques that could help in various case studies.

Module 3, Advanced Machine Learning with Python, helps you to build your skill with deep architectures by using stacked denoising autoencoders. This module is a blend of semi-supervised learning techniques, RBM and DBN algorithms .Further this focuses on tools and techniques which will help in making consistent working process.

What you need for this learning path

Module 1, Python Machine Learning will require an installation of Python 3.4.3 or newer on Mac OS X, Linux or Microsoft Windows. Use of Python essential libraries like SciPy, NumPy, scikit-Learn, matplotlib, and pandas. is essential.

Before you start, Please refer:

- The direct link to the Iris dataset would be: `https://raw.githubusercontent.com/rasbt/python-machine-learning-book/master/code/datasets/iris/iris.data`

- We've added some additional notes to the code notebooks mentioning the offline datasets in case there are server errors. `https://www.dropbox.com/sh/tq2qdh0oqfgsktq/AADIt7esnbiWLOQODn5q_7Dta?dl=0`

- *Module 2, Designing Machine Learning Systems with Python*, will need an inclination to learn machine learning and the Python V3 software, which you can download from `https://www.python.org/downloads/`.

- Module 3, Advanced Machine Learning with Python, leverages openly available data and code, including open source Python libraries and frameworks.

Who this learning path is for

This title is for Data scientist and researchers who are already into the field of Data Science and want to see Machine learning in action and explore its real-world application. Prior knowledge of Python programming and mathematics is must with basic knowledge of machine learning concepts.

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/Python-Deeper-Insights-into-Machine-Learning`.

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.

Course Module 1: Python Machine Learning

Course Module 3: Advanced Machine Learning with Python

Module 1

Python Machine Learning

Leverage benefits of machine learning techniques using Python

1
Giving Computers the Ability to Learn from Data

In my opinion, *machine learning*, the application and science of algorithms that makes sense of data, is the most exciting field of all the computer sciences! We are living in an age where data comes in abundance; using the self-learning algorithms from the field of machine learning, we can turn this data into knowledge. Thanks to the many powerful open source libraries that have been developed in recent years, there has probably never been a better time to break into the machine learning field and learn how to utilize powerful algorithms to spot patterns in data and make predictions about future events.

In this chapter, we will learn about the main concepts and different types of machine learning. Together with a basic introduction to the relevant terminology, we will lay the groundwork for successfully using machine learning techniques for practical problem solving.

In this chapter, we will cover the following topics:
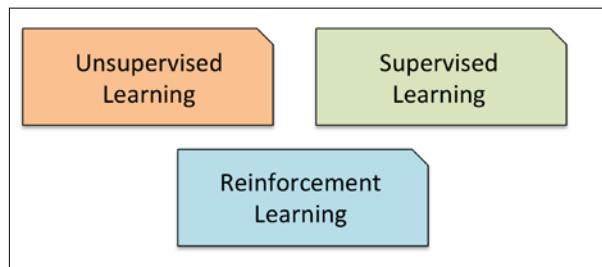
- The general concepts of machine learning
- The three types of learning and basic terminology
- The building blocks for successfully designing machine learning systems
- Installing and setting up Python for data analysis and machine learning

Building intelligent machines to transform data into knowledge

In this age of modern technology, there is one resource that we have in abundance: a large amount of structured and unstructured data. In the second half of the twentieth century, machine learning evolved as a subfield of *artificial intelligence* that involved the development of self-learning algorithms to gain knowledge from that data in order to make predictions. Instead of requiring humans to manually derive rules and build models from analyzing large amounts of data, machine learning offers a more efficient alternative for capturing the knowledge in data to gradually improve the performance of predictive models, and make data-driven decisions. Not only is machine learning becoming increasingly important in computer science research but it also plays an ever greater role in our everyday life. Thanks to machine learning, we enjoy robust e-mail spam filters, convenient text and voice recognition software, reliable Web search engines, challenging chess players, and, hopefully soon, safe and efficient self-driving cars.

The three different types of machine learning
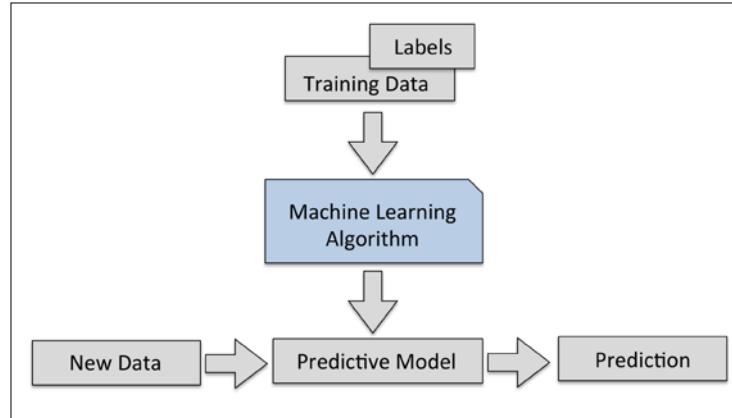
In this section, we will take a look at the three types of machine learning: *supervised learning*, *unsupervised learning*, and *reinforcement learning*. We will learn about the fundamental differences between the three different learning types and, using conceptual examples, we will develop an intuition for the practical problem domains where these can be applied:

Making predictions about the future with supervised learning

The main goal in supervised learning is to learn a model from labeled *training* data that allows us to make predictions about unseen or future data. Here, the term *supervised* refers to a set of samples where the desired output signals (labels) are already known.

Considering the example of e-mail spam filtering, we can train a model using a supervised machine learning algorithm on a corpus of labeled e-mail, e-mail that are correctly marked as spam or not-spam, to predict whether a new e-mail belongs to either of the two categories. A supervised learning task with discrete *class labels*, such as in the previous e-mail spam-filtering example, is also called a *classification* task. Another subcategory of supervised learning is *regression*, where the outcome signal is a continuous value:
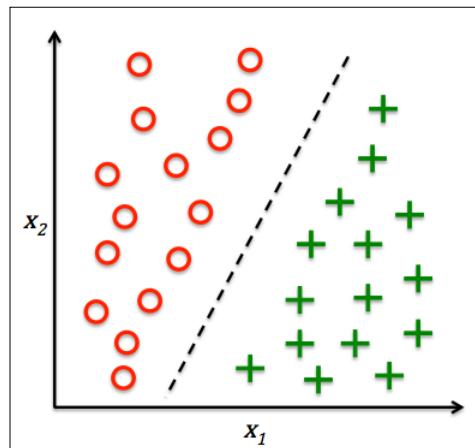
Classification for predicting class labels

Classification is a subcategory of supervised learning where the goal is to predict the categorical class labels of new instances based on past observations. Those class labels are discrete, unordered values that can be understood as the *group memberships* of the instances. The previously mentioned example of e-mail-spam detection represents a typical example of a *binary classification* task, where the machine learning algorithm learns a set of rules in order to distinguish between two possible classes: spam and non-spam e-mail.

However, the set of class labels does not have to be of a binary nature. The predictive model learned by a supervised learning algorithm can assign any class label that was presented in the training dataset to a new, unlabeled instance. A typical example of a *multi-class classification* task is handwritten character recognition. Here, we could collect a training dataset that consists of multiple handwritten examples of each letter in the alphabet. Now, if a user provides a new handwritten character via an input device, our predictive model will be able to predict the correct letter in the alphabet with certain accuracy. However, our machine learning system would be unable to correctly recognize any of the digits zero to nine, for example, if they were not part of our training dataset.

The following figure illustrates the concept of a binary classification task given 30 training samples: 15 training samples are labeled as *negative class* (circles) and 15 training samples are labeled as *positive class* (plus signs). In this scenario, our dataset is two-dimensional, which means that each sample has two values associated with it: x_1 and x_2. Now, we can use a supervised machine learning algorithm to learn a rule—the decision boundary represented as a black dashed line—that can separate those two classes and classify new data into each of those two categories given its x_1 and x_2 values:

Regression for predicting continuous outcomes
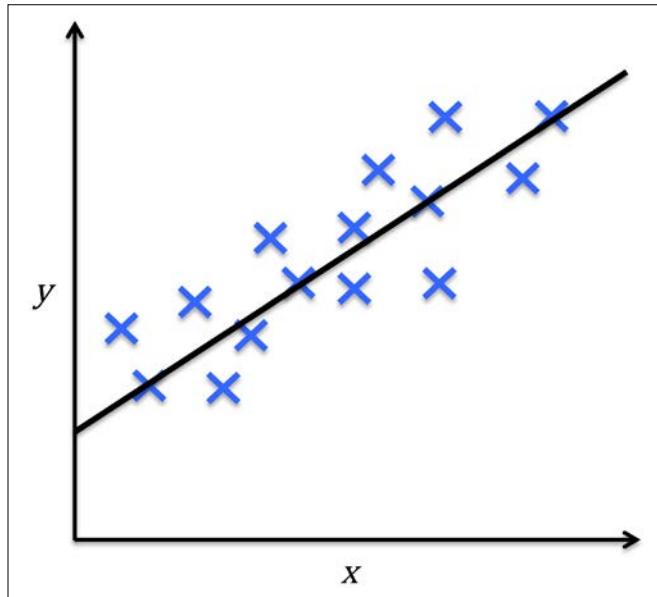
We learned in the previous section that the task of classification is to assign categorical, unordered labels to instances. A second type of supervised learning is the prediction of continuous outcomes, which is also called regression analysis. In *regression analysis*, we are given a number of *predictor* (explanatory) variables and a continuous response variable (outcome), and we try to find a relationship between those variables that allows us to predict an outcome.

For example, let's assume that we are interested in predicting the Math SAT scores of our students. If there is a relationship between the time spent studying for the test and the final scores, we could use it as training data to learn a model that uses the study time to predict the test scores of future students who are planning to take this test.

> The term *regression* was devised by Francis Galton in his article *Regression Towards Mediocrity in Hereditary Stature* in 1886. Galton described the biological phenomenon that the variance of *height* in a population does not increase over time. He observed that the height of parents is not passed on to their children but the children's height is regressing towards the population mean.
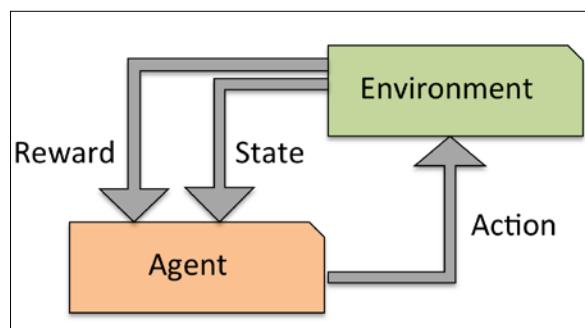
The following figure illustrates the concept of *linear regression*. Given a predictor variable *x* and a response variable *y*, we fit a straight line to this data that minimizes the distance—most commonly the average squared distance—between the sample points and the fitted line. We can now use the intercept and slope learned from this data to predict the outcome variable of new data:

Solving interactive problems with reinforcement learning

Another type of machine learning is reinforcement learning. In reinforcement learning, the goal is to develop a system (*agent*) that improves its performance based on interactions with the *environment*. Since the information about the current state of the environment typically also includes a so-called *reward* signal, we can think of reinforcement learning as a field related to *supervised* learning. However, in reinforcement learning this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a *reward* function. Through the interaction with the environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward via an exploratory trial-and-error approach or deliberative planning.

A popular example of reinforcement learning is a chess engine. Here, the agent decides upon a series of moves depending on the state of the board (the environment), and the reward can be defined as *win* or *lose* at the end of the game:
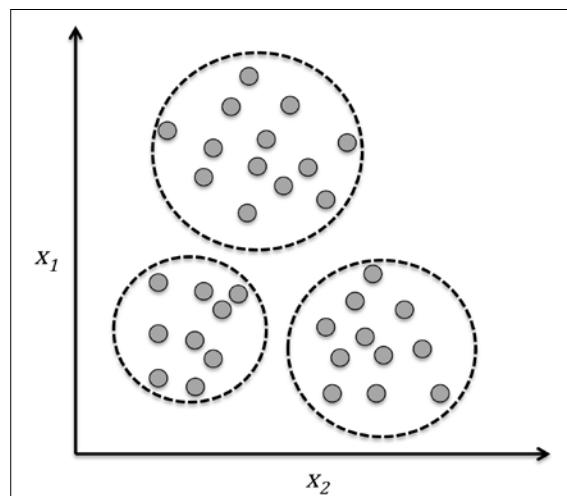
Discovering hidden structures with unsupervised learning

In supervised learning, we know the *right answer* beforehand when we train our model, and in reinforcement learning, we define a measure of *reward* for particular actions by the agent. In unsupervised learning, however, we are dealing with unlabeled data or data of *unknown structure*. Using unsupervised learning techniques, we are able to explore the structure of our data to extract meaningful information without the guidance of a known outcome variable or reward function.

Finding subgroups with clustering

Clustering is an exploratory data analysis technique that allows us to organize a pile of information into meaningful subgroups (*clusters*) without having any prior knowledge of their group memberships. Each cluster that may arise during the analysis defines a group of objects that share a certain degree of similarity but are more dissimilar to objects in other clusters, which is why clustering is also sometimes called "unsupervised classification." Clustering is a great technique for structuring information and deriving meaningful relationships among data, For example, it allows marketers to discover customer groups based on their interests in order to develop distinct marketing programs.
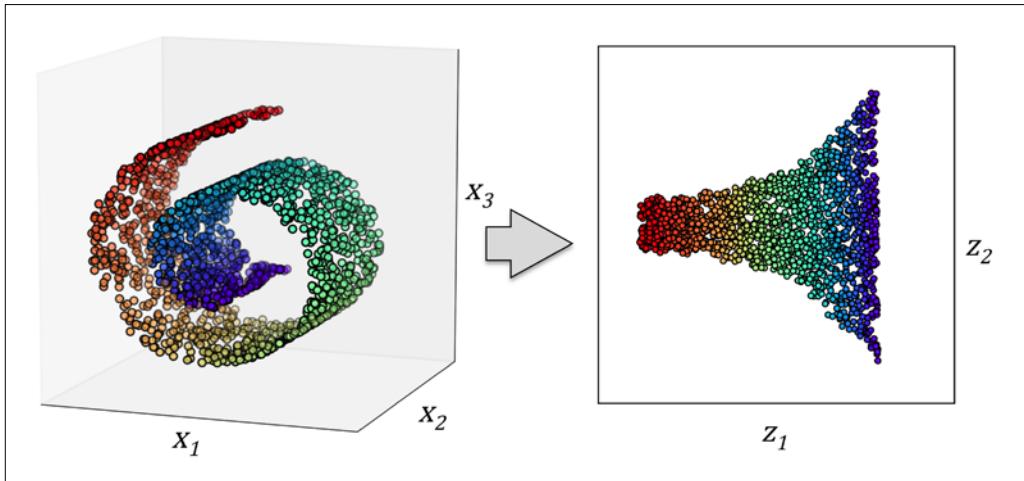
The figure below illustrates how clustering can be applied to organizing unlabeled data into three distinct groups based on the similarity of their features x_1 and x_2:

Dimensionality reduction for data compression
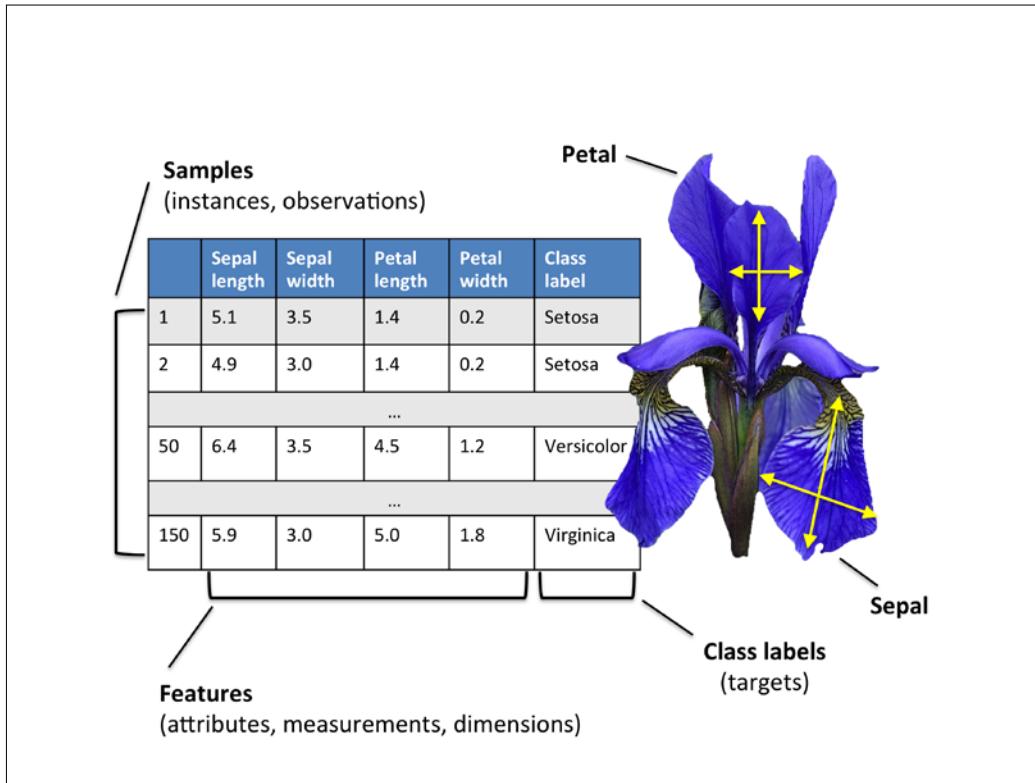
Another subfield of unsupervised learning is *dimensionality reduction*. Often we are working with data of high dimensionality—each observation comes with a high number of measurements—that can present a challenge for limited storage space and the computational performance of machine learning algorithms. Unsupervised dimensionality reduction is a commonly used approach in feature preprocessing to remove noise from data, which can also degrade the predictive performance of certain algorithms, and compress the data onto a smaller dimensional subspace while retaining most of the relevant information.

Sometimes, dimensionality reduction can also be useful for visualizing data—for example, a high-dimensional feature set can be projected onto one-, two-, or three-dimensional feature spaces in order to visualize it via 3D- or 2D-scatterplots or histograms. The figure below shows an example where non-linear dimensionality reduction was applied to compress a 3D *Swiss Roll* onto a new 2D feature subspace:

An introduction to the basic terminology and notations

Now that we have discussed the three broad categories of machine learning—supervised, unsupervised, and reinforcement learning—let us have a look at the basic terminology that we will be using in the next chapters. The following table depicts an excerpt of the *Iris* dataset, which is a classic example in the field of machine learning. The Iris dataset contains the measurements of 150 iris flowers from three different species: *Setosa*, *Versicolor*, and *Virginica*. Please check if this is replaced. Here, each flower sample represents one row in our data set, and the flower measurements in centimeters are stored as columns, which we also call the features of the dataset:

To keep the notation and implementation simple yet efficient, we will make use of some of the basics of *linear algebra*. In the following chapters, we will use a *matrix* and *vector* notation to refer to our data. We will follow the common convention to represent each sample as separate row in a feature matrix X, where each feature is stored as a separate column.

The Iris dataset, consisting of 150 samples and 4 features, can then be written as a 150×4 matrix $X \in \mathbb{R}^{150 \times 4}$:

$$
\begin{bmatrix}
x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\
x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\
\vdots & \vdots & \vdots & \vdots \\
x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)}
\end{bmatrix}
$$

For the rest of this book, we will use the superscript *(i)* to refer to the *i*th training sample, and the subscript *j* to refer to the *j*th dimension of the training dataset.

We use lower-case, bold-face letters to refer to vectors $\left(x \in \mathbb{R}^{n \times 1} \right)$ and upper-case, bold-face letters to refer to matrices, respectively $(X \in \mathbb{R}^{n \times m})$. To refer to single elements in a vector or matrix, we write the letters in italics ($x^{(n)}$ or $x_{(m)}^{(n)}$, respectively).

For example, x_1^{150} refers to the first dimension of flower sample 150, the *sepal length*. Thus, each row in this feature matrix represents one flower instance and can be written as four-dimensional row vector $x^{(i)} \in \mathbb{R}^{1 \times 4}$,

$$ x^{(i)} = \left[x_1^{(i)} \quad x_2^{(i)} \quad x_3^{(i)} \quad x_4^{(i)} \right]. $$

Each feature dimension is a 150-dimensional column vector $x_j \in \mathbb{R}^{150 \times 1}$, for example:
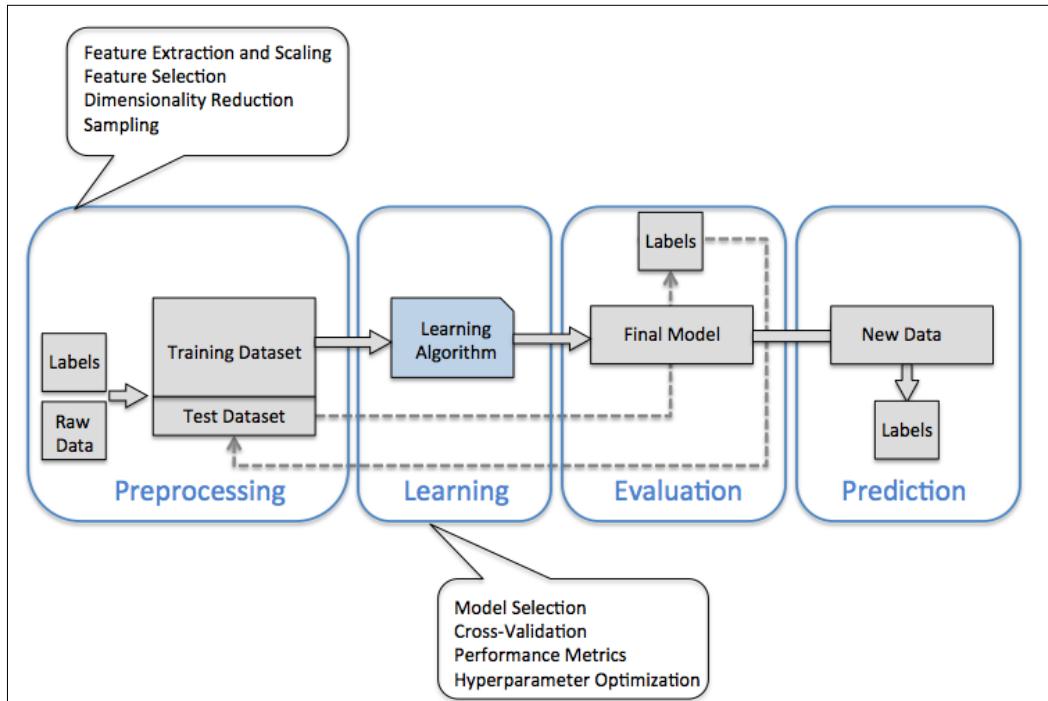
$$ x_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}. $$

Similarly, we store the target variables (here: class labels) as a 150-dimensional column vector $y = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} \left(y \in \{ \text{Setosa, Versicolor, Virginica} \} \right)$.

A roadmap for building machine learning systems

In the previous sections, we discussed the basic concepts of machine learning and the three different types of learning. In this section, we will discuss other important parts of a machine learning system accompanying the learning algorithm. The diagram below shows a typical workflow diagram for using machine learning in *predictive modeling*, which we will discuss in the following subsections:

Preprocessing – getting data into shape

Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm. Thus, the *preprocessing* of the data is one of the most crucial steps in any machine learning application. If we take the Iris flower dataset from the previous section as an example, we could think of the raw data as a series of flower images from which we want to extract meaningful features. Useful features could be the color, the hue, the intensity of the flowers, the height, and the flower lengths and widths. Many machine learning algorithms also require that the selected features are on the same scale for optimal performance, which is often achieved by transforming the features in the range [0, 1] or a standard normal distribution with zero mean and unit variance, as we will see in the later chapters.

Some of the selected features may be highly correlated and therefore redundant to a certain degree. In those cases, dimensionality reduction techniques are useful for compressing the features onto a lower dimensional subspace. Reducing the dimensionality of our feature space has the advantage that less storage space is required, and the learning algorithm can run much faster.

To determine whether our machine learning algorithm not only performs well on the training set but also generalizes well to new data, we also want to randomly divide the dataset into a separate training and test set. We use the training set to train and optimize our machine learning model, while we keep the test set until the very end to evaluate the final model.

Training and selecting a predictive model

As we will see in later chapters, many different machine learning algorithms have been developed to solve different problem tasks. An important point that can be summarized from David Wolpert's famous *No Free Lunch Theorems* is that we can't get learning "for free" (*The Lack of A Priori Distinctions Between Learning Algorithms*, D.H. Wolpert 1996; *No Free Lunch Theorems for Optimization*, D.H. Wolpert and W.G. Macready, 1997). Intuitively, we can relate this concept to the popular saying, "*I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail*" (Abraham Maslow, 1966). For example, each classification algorithm has its inherent biases, and no single classification model enjoys superiority if we don't make any assumptions about the task. In practice, it is therefore essential to compare at least a handful of different algorithms in order to train and select the best performing model. But before we can compare different models, we first have to decide upon a metric to measure performance. One commonly used metric is classification accuracy, which is defined as the proportion of correctly classified instances.

One legitimate question to ask is: *how do we know which model performs well on the final test dataset and real-world data if we don't use this test set for the model selection but keep it for the final model evaluation?* In order to address the issue embedded in this question, different cross-validation techniques can be used where the training dataset is further divided into training and *validation subsets* in order to estimate the *generalization performance* of the model. Finally, we also cannot expect that the default parameters of the different learning algorithms provided by software libraries are optimal for our specific problem task. Therefore, we will make frequent use of hyperparameter *optimization techniques* that help us to fine-tune the performance of our model in later chapters. Intuitively, we can think of those hyperparameters as parameters that are not learned from the data but represent the knobs of a model that we can turn to improve its performance, which will become much clearer in later chapters when we see actual examples.

Evaluating models and predicting unseen data instances

After we have selected a model that has been fitted on the training dataset, we can use the test dataset to estimate how well it performs on this unseen data to estimate the generalization error. If we are satisfied with its performance, we can now use this model to predict new, future data. It is important to note that the parameters for the previously mentioned procedures—such as feature scaling and dimensionality reduction—are solely obtained from the training dataset, and the same parameters are later re-applied to transform the test dataset, as well as any new data samples—the performance measured on the test data may be overoptimistic otherwise.

Using Python for machine learning

Python is one of the most popular programming languages for data science and therefore enjoys a large number of useful add-on libraries developed by its great community.

Although the performance of interpreted languages, such as Python, for computation-intensive tasks is inferior to lower-level programming languages, extension libraries such as *NumPy* and *SciPy* have been developed that build upon lower layer Fortran and C implementations for fast and vectorized operations on multidimensional arrays.

For machine learning programming tasks, we will mostly refer to the *scikit-learn* library, which is one of the most popular and accessible open source machine learning libraries as of today.

Installing Python packages

Python is available for all three major operating systems—Microsoft Windows, Mac OS X, and Linux—and the installer, as well as the documentation, can be downloaded from the official Python website: `https://www.python.org`.

This book is written for Python version >= 3.4.3, and it is recommended you use the most recent version of Python 3 that is currently available, although most of the code examples may also be compatible with Python >= 2.7.10. If you decide to use Python 2.7 to execute the code examples, please make sure that you know about the major differences between the two Python versions. A good summary about the differences between Python 3.4 and 2.7 can be found at `https://wiki.python.org/moin/Python2orPython3`.

The additional packages that we will be using throughout this book can be installed via the *pip* installer program, which has been part of the Python standard library since Python 3.3. More information about pip can be found at `https://docs.python.org/3/installing/index.html`.

After we have successfully installed Python, we can execute pip from the command line terminal to install additional Python packages:

```
pip install SomePackage
```

Already installed packages can be updated via the `--upgrade` flag:

```
pip install SomePackage --upgrade
```

A highly recommended alternative Python distribution for scientific computing is Anaconda by Continuum Analytics. Anaconda is a free—including commercial use—enterprise-ready Python distribution that bundles all the essential Python packages for data science, math, and engineering in one user-friendly cross-platform distribution. The Anaconda installer can be downloaded at `http://continuum.io/downloads#py34`, and an Anaconda quick start-guide is available at `https://store.continuum.io/static/img/Anaconda-Quickstart.pdf`.

After successfully installing Anaconda, we can install new Python packages using the following command:

```
conda install SomePackage
```

Existing packages can be updated using the following command:

```
conda update SomePackage
```

Throughout this book, we will mainly use *NumPy*'s multi-dimensional arrays to store and manipulate data. Occasionally, we will make use of *pandas*, which is a library built on top of NumPy that provides additional higher level data manipulation tools that make working with tabular data even more convenient. To augment our learning experience and visualize quantitative data, which is often extremely useful to intuitively make sense of it, we will use the very customizable *matplotlib* library.

The version numbers of the major Python packages that were used for writing this book are listed below. Please make sure that the version numbers of your installed packages are equal to, or greater than, those version numbers to ensure the code examples run correctly:

- NumPy 1.9.1
- SciPy 0.14.0
- scikit-learn 0.15.2
- matplotlib 1.4.0
- pandas 0.15.2

Summary

In this chapter, we explored machine learning on a very high level and familiarized ourselves with the big picture and major concepts that we are going to explore in the next chapters in more detail.

We learned that supervised learning is composed of two important subfields: classification and regression. While classification models allow us to categorize objects into known classes, we can use regression analysis to predict the continuous outcomes of target variables. Unsupervised learning not only offers useful techniques for discovering structures in unlabeled data, but it can also be useful for data compression in feature preprocessing steps.

We briefly went over the typical roadmap for applying machine learning to problem tasks, which we will use as a foundation for deeper discussions and hands-on examples in the following chapters. Eventually, we set up our Python environment and installed and updated the required packages to get ready to see machine-learning in action.

In the following chapter, we will implement one of the earliest machine learning algorithms for classification that will prepare us for *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, where we cover more advanced machine learning algorithms using the scikit-learn open source machine learning library. Since machine learning algorithms learn from data, it is critical that we feed them useful information, and in *Chapter 4*, *Building Good Training Sets – Data Preprocessing* we will take a look at important data preprocessing techniques. In *Chapter 5*, *Compressing Data via Dimensionality Reduction*, we will learn about dimensionality reduction techniques that can help us to compress our dataset onto a lower-dimensional feature subspace, which can be beneficial for computational efficiency. An important aspect of building machine learning models is to evaluate their performance and to estimate how well they can make predictions on new, unseen data. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning* we will learn all about the best practices for model tuning and evaluation. In certain scenarios, we still may not be satisfied with the performance of our predictive model although we may have spent hours or days extensively tuning and testing. In *Chapter 7*, *Combining Different Models for Ensemble Learning* we will learn how to combine different machine learning models to build even more powerful predictive systems.

After we covered all of the important concepts of a typical machine learning pipeline, we will implement a model for predicting emotions in text in *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*, and in *Chapter 9*, *Embedding a Machine Learning Model into a Web Application*, we will embed it into a Web application to share it with the world. In *Chapter 10*, *Predicting Continuous Target Variables with Regression Analysis* we will then use machine learning algorithms for regression analysis that allow us to predict continuous output variables, and in *Chapter 11*, *Working with Unlabelled Data – Clustering Analysis* we will apply clustering algorithms that will allow us to find hidden structures in data. The last two chapters in this book will cover artificial neural networks that will allow us to tackle complex problems, such as image and speech recognition, which is currently one of the hottest topics in machine-learning research.

2
Training Machine Learning Algorithms for Classification
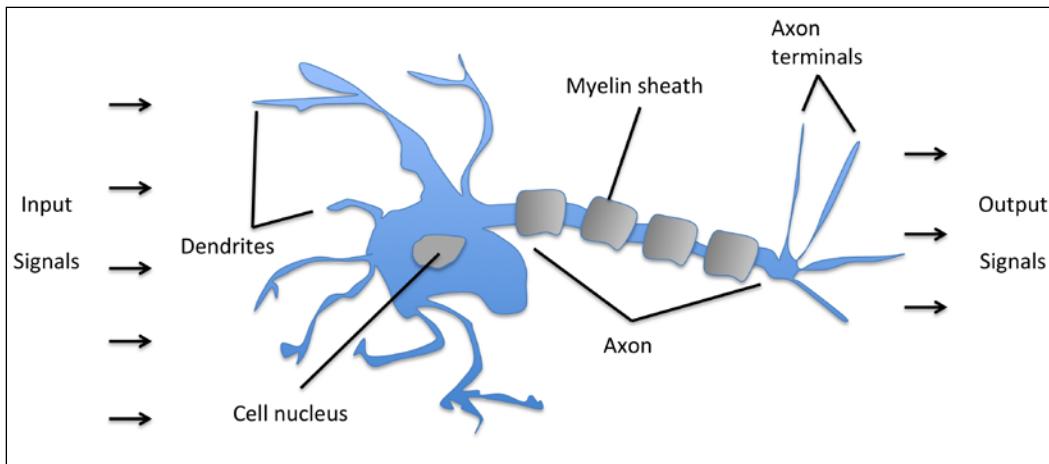
In this chapter, we will make use of one of the first algorithmically described machine learning algorithms for classification, the *perceptron* and *adaptive linear neurons*. We will start by implementing a perceptron step by step in Python and training it to classify different flower species in the Iris dataset. This will help us to understand the concept of machine learning algorithms for classification and how they can be efficiently implemented in Python. Discussing the basics of optimization using adaptive linear neurons will then lay the groundwork for using more powerful classifiers via the scikit-learn machine-learning library in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*.

The topics that we will cover in this chapter are as follows:

- Building an intuition for machine learning algorithms
- Using pandas, NumPy, and matplotlib to read in, process, and visualize data
- Implementing linear classification algorithms in Python

Artificial neurons – a brief glimpse into the early history of machine learning

Before we discuss the perceptron and related algorithms in more detail, let us take a brief tour through the early beginnings of machine learning. Trying to understand how the biological brain works to design artificial intelligence, Warren McCullock and Walter Pitts published the first concept of a simplified brain cell, the so-called *McCullock-Pitts (MCP) neuron*, in 1943 (W. S. McCulloch and W. Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity*. The bulletin of mathematical biophysics, 5(4):115–133, 1943). Neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in the following figure:

McCullock and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton.* Cornell Aeronautical Laboratory, 1957). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that are then multiplied with the input features in order to make the decision of whether a neuron fires or not. In the context of supervised learning and classification, such an algorithm could then be used to predict if a sample belonged to one class or the other.

More formally, we can pose this problem as a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. We can then define an *activation function* $\phi(z)$ that takes a linear combination of certain input values x and a corresponding weight vector w, where z is the so-called net input ($z = w_1 x_1 + \ldots + w_m x_m$):

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \ x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Now, if the activation of a particular sample $x^{(i)}$, that is, the output of $\phi(z)$, is greater than a defined threshold θ, we predict class 1 and class -1, otherwise. In the perceptron algorithm, the activation function $\phi(\cdot)$ is a simple *unit step function*, which is sometimes also called the *Heaviside step function*:

$$\phi(z) = \begin{cases} 1 & if \ z \geq \theta \\ -1 & otherwise \end{cases}$$

For simplicity, we can bring the threshold θ to the left side of the equation and define a weight-zero as $w_0 = -\theta$ and $x_0 = 1$, so that we write $\mathbf{z}$ in a more compact form $z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \mathbf{w}^T \mathbf{x}$ and $\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$.

> In the following sections, we will often make use of basic notations from linear algebra. For example, we will abbreviate the sum of the products of the values in $\mathbf{x}$ and $\mathbf{w}$ using a *vector dot product*, whereas superscript **T** stands for *transpose*, which is an operation that transforms a column vector into a row vector and vice versa:
>
> $$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \mathbf{w}^T \mathbf{x}$$
>
> For example: $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$.
>
> Furthermore, the transpose operation can also be applied to a matrix to reflect it over its diagonal, for example:
>
> $$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$
>
> In this book, we will only use the very basic concepts from linear algebra. However, if you need a quick refresher, please take a look at Zico Kolter's excellent Linear Algebra Review and Reference, which is freely available at http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf.
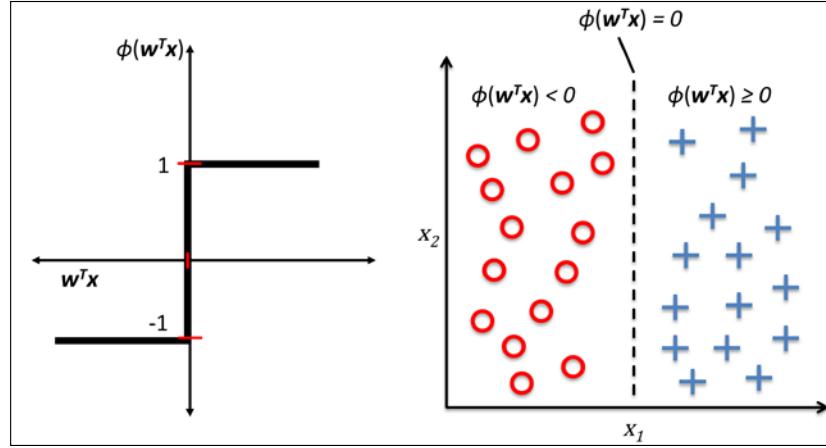
The following figure illustrates how the net input $z = \mathbf{w}^T \mathbf{x}$ is squashed into a binary output (-1 or 1) by the activation function of the perceptron (left subfigure) and how it can be used to discriminate between two linearly separable classes (right subfigure):

The whole idea behind the MCP neuron and Rosenblatt's *thresholded* perceptron model is to use a reductionist approach to mimic how a single neuron in the brain works: it either *fires* or it doesn't. Thus, Rosenblatt's initial perceptron rule is fairly simple and can be summarized by the following steps:

1. Initialize the weights to 0 or small random numbers.

2. For each training sample $x^{(i)}$ perform the following steps:

    1. Compute the output value $\hat{y}$.
    2. Update the weights.

Here, the output value is the class label predicted by the unit step function that we defined earlier, and the simultaneous update of each weight w_j in the weight vector w can be more formally written as:

$$w_j := w_j + \Delta w_j$$

The value of Δw_j, which is used to update the weight w_j, is calculated by the perceptron learning rule:

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

Where η is the learning rate (a constant between 0.0 and 1.0), $y^{(i)}$ is the true class label of the ith training sample, and $\hat{y}^{(i)}$ is the predicted class label. It is important to note that all weights in the weight vector are being updated simultaneously, which means that we don't recompute the $\hat{y}^{(i)}$ before all of the weights Δw_j were updated. Concretely, for a 2D dataset, we would write the update as follows:

$$\Delta w_0 = \eta \left(y^{(i)} - output^{(i)} \right)$$

$$\Delta w_1 = \eta \left(y^{(i)} - output^{(i)} \right) x_1^{(i)}$$

$$\Delta w_2 = \eta \left(y^{(i)} - output^{(i)} \right) x_2^{(i)}$$

Before we implement the perceptron rule in Python, let us make a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the weights remain unchanged:

$$\Delta w_j = \eta \left(-1 - -1 \right) x_j^{(i)} = 0$$

$$\Delta w_j = \eta \left(1 - 1 \right) x_j^{(i)} = 0$$

However, in the case of a wrong prediction, the weights are being pushed towards the direction of the positive or negative target class, respectively:

$$\Delta w_j = \eta \left(1 - -1 \right) x_j^{(i)} = \eta \left(2 \right) x_j^{(i)}$$

$$\Delta w_j = \eta \left(-1 - 1 \right) x_j^{(i)} = \eta \left(-2 \right) x_j^{(i)}$$

To get a better intuition for the multiplicative factor $x_j^{(i)}$, let us go through another simple example, where:

$$y^{(i)} = +1, \ \hat{y}_j^{(i)} = -1, \ \eta = 1$$
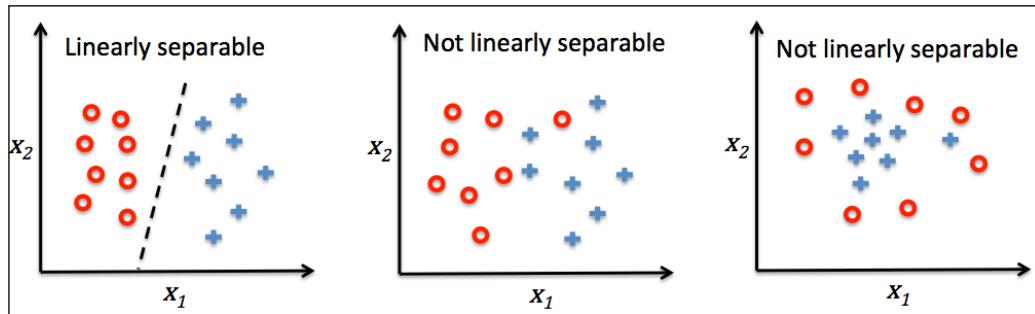
Let's assume that $x_j^{(i)} = 0.5$, and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1 so that the activation $x_j^{(i)} \times w_j^{(i)}$ will be more positive the next time we encounter this sample and thus will be more likely to be above the threshold of the unit step function to classify the sample as +1:

$$\Delta w_j^{(i)} = \left(1 - -1\right)0.5 = \left(2\right)0.5 = 1$$

The weight update is proportional to the value of $x_j^{(i)}$. For example, if we have another sample $x_j^{(i)} = 2$ that is incorrectly classified as -1, we'd push the decision boundary by an even larger extent to classify this sample correctly the next time:

$$\Delta w_j = \left(1 - -1\right)2 = \left(2\right)2 = 4$$

It is important to note that the convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small. If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (*epochs*) and/or a threshold for the number of tolerated misclassifications—the perceptron would never stop updating the weights otherwise:
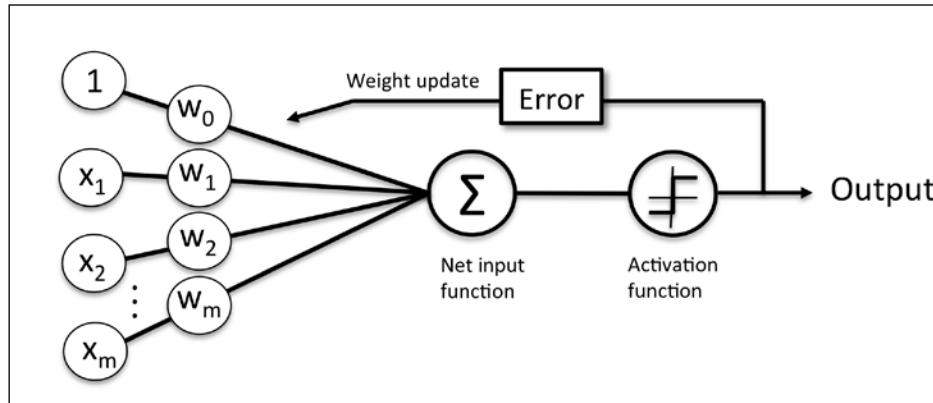
Now, before we jump into the implementation in the next section, let us summarize what we just learned in a simple figure that illustrates the general concept of the perceptron:

The preceding figure illustrates how the perceptron receives the inputs of a sample x and combines them with the weights w to compute the net input. The net input is then passed on to the activation function (here: the unit step function), which generates a binary output -1 or +1—the predicted class label of the sample. During the learning phase, this output is used to calculate the error of the prediction and update the weights.

Implementing a perceptron learning algorithm in Python

In the previous section, we learned how Rosenblatt's perceptron rule works; let us now go ahead and implement it in Python and apply it to the Iris dataset that we introduced in *Chapter 1*, *Giving Computers the Ability to Learn from Data*. We will take an objected-oriented approach to define the perceptron interface as a Python `Class`, which allows us to initialize new perceptron objects that can learn from data via a `fit` method, and make predictions via a separate `predict` method. As a convention, we add an underscore to attributes that are not being created upon the initialization of the object but by calling the object's other methods—for example, `self.w_`.

> If you are not yet familiar with Python's scientific libraries or need a
> refresher, please see the following resources:
>
> **NumPy**: `http://wiki.scipy.org/Tentative_NumPy_Tutorial`
>
> **Pandas**: `http://pandas.pydata.org/pandas-docs/stable/ tutorials.html`
>
> **Matplotlib**: `http://matplotlib.org/users/beginner.html`
>
> Also, to better follow the code examples, I recommend you download
> the IPython notebooks from the Packt website. For a general
> introduction to IPython notebooks, please visit `https://ipython. org/ipython-doc/3/notebook/index.html`.

```python
import numpy as np
class Perceptron(object):
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.

    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples
            is the number of samples and
```

```
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```
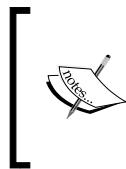
Using this perceptron implementation, we can now initialize new `Perceptron` objects with a given learning rate `eta` and `n_iter`, which is the number of epochs (passes over the training set). Via the `fit` method we initialize the weights in `self.w_` to a zero-vector $\mathbb{R}^{m+1}$ where m stands for the number of dimensions (features) in the dataset where we add 1 for the zero-weight (that is, the threshold).

> NumPy indexing for one-dimensional arrays works similarly to Python lists using the square-bracket (`[]`) notation. For two-dimensional arrays, the first indexer refers to the row number, and the second indexer to the column number. For example, we would use `X[2, 3]` to select the third row and fourth column of a 2D array `X`.

After the weights have been initialized, the `fit` method loops over all individual samples in the training set and updates the weights according to the perceptron learning rule that we discussed in the previous section. The class labels are predicted by the `predict` method, which is also called in the `fit` method to predict the class label for the weight update, but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the list `self.errors_` so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product $w^T x$.

> Instead of using NumPy to calculate the vector dot product between two arrays a and b via `a.dot(b)` or `np.dot(a, b)`, we could also perform the calculation in pure Python via `sum([i*j for i,j in zip(a, b)])`. However, the advantage of using NumPy over classic Python for-loop structures is that its arithmetic operations are vectorized. **Vectorization** means that an elemental arithmetic operation is automatically applied to all elements in an array. By formulating our arithmetic operations as a sequence of instructions on an array rather than performing a set of operations for each element one at a time, we can make better use of our modern CPU architectures with **Single Instruction, Multiple Data (SIMD)** support. Furthermore, NumPy uses highly optimized linear algebra libraries, such as **Basic Linear Algebra Subprograms (BLAS)** and **Linear Algebra Package (LAPACK)** that have been written in C or Fortran. Lastly, NumPy also allows us to write our code in a more compact and intuitive way using the basics of linear algebra, such as vector and matrix dot products.

Training a perceptron model on the Iris dataset

To test our perceptron implementation, we will load the two flower classes *Setosa* and *Versicolor* from the Iris dataset. Although, the perceptron rule is not restricted to two dimensions, we will only consider the two features *sepal length* and *petal length* for visualization purposes. Also, we only chose the two flower classes *Setosa* and *Versicolor* for practical reasons. However, the perceptron algorithm can be extended to multi-class classification—for example, through the *One-vs.-All* technique.

> **One-vs.-All (OvA)**, or sometimes also called **One-vs.-Rest (OvR)**, is a technique used to extend a binary classifier to multi-class problems. Using OvA, we can train one classifier per class, where the particular class is treated as the positive class and the samples from all other classes are considered as the negative class. If we were to classify a new data sample, we would use our n classifiers, where n is the number of class labels, and assign the class label with the highest confidence to the particular sample. In the case of the perceptron, we would use OvA to choose the class label that is associated with the largest absolute net input value.

First, we will use the *pandas* library to load the Iris dataset directly from the *UCI Machine Learning Repository* into a `DataFrame` object and print the last five lines via the `tail` method to check that the data was loaded correctly:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...    'machine-learning-databases/iris/iris.data', header=None)
>>> df.tail()
```

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Next, we extract the first 100 class labels that correspond to the 50 *Iris-Setosa* and 50 *Iris-Versicolor* flowers, respectively, and convert the class labels into the two integer class labels `1` (*Versicolor*) and `-1` (*Setosa*) that we assign to a vector `y` where the values method of a pandas `DataFrame` yields the corresponding NumPy representation. Similarly, we extract the first feature column (*sepal length*) and the third feature column (*petal length*) of those 100 training samples and assign them to a feature matrix `X`, which we can visualize via a two-dimensional scatter plot:
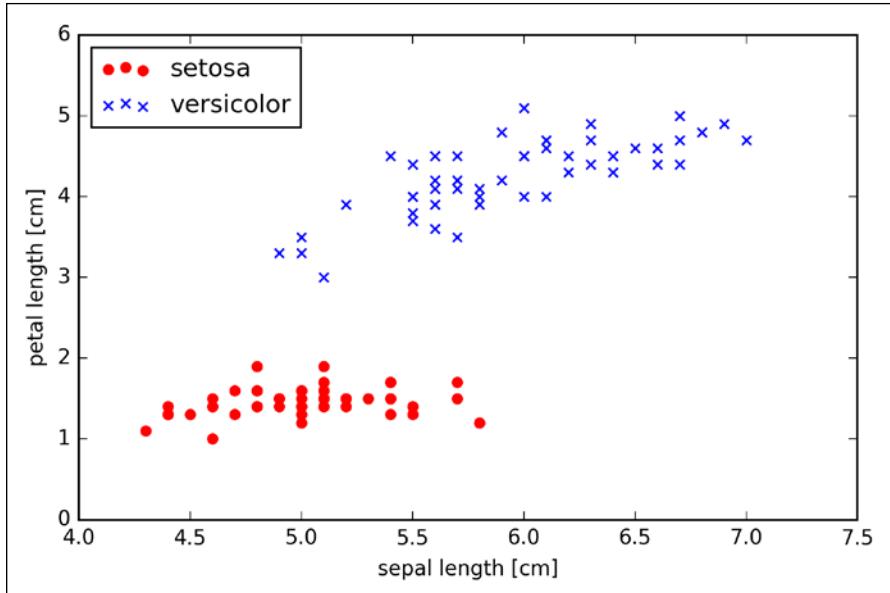
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np

>>> y = df.iloc[0:100, 4].values
```

```
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> X = df.iloc[0:100, [0, 2]].values
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='o', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='x', label='versicolor')
>>> plt.xlabel('sepal length')
>>> plt.ylabel('petal length')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example we should now see the following scatterplot:

Now it's time to train our perceptron algorithm on the Iris data subset that we just extracted. Also, we will plot the `misclassification error` for each epoch to check if the algorithm converged and found a decision boundary that separates the two Iris flower classes:
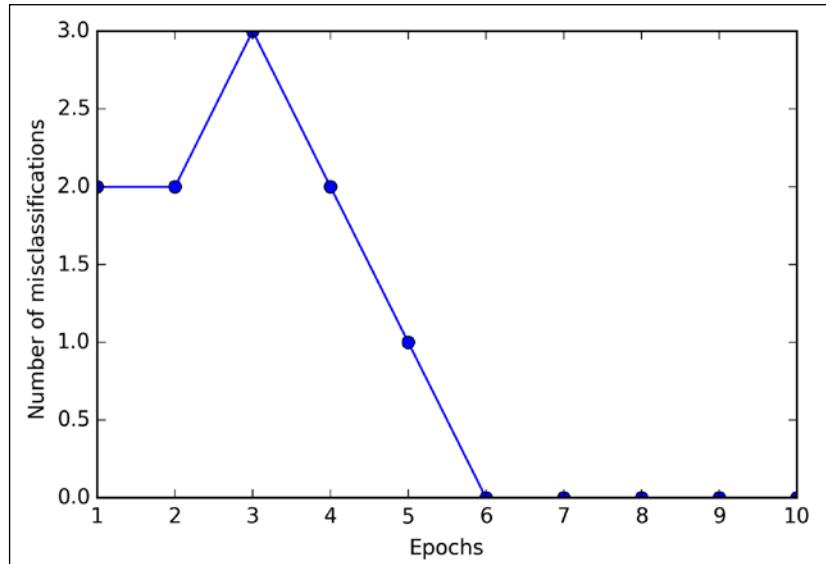
```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_,
```

```
...              marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Number of misclassifications')
>>> plt.show()
```

After executing the preceding code, we should see the plot of the misclassification errors versus the number of epochs, as shown next:

As we can see in the preceding plot, our perceptron already converged after the sixth epoch and should now be able to classify the training samples perfectly. Let us implement a small convenience function to visualize the decision boundaries for 2D datasets:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
```

```
cmap = ListedColormap(colors[:len(np.unique(y))])


# plot the decision surface
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                       np.arange(x2_min, x2_max, resolution))
Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())


# plot class samples
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                alpha=0.8, c=cmap(idx),
                marker=markers[idx], label=cl)
```

First, we define a number of `colors` and `markers` and create a color map from the list of colors via `ListedColormap`. Then, we determine the minimum and maximum values for the two features and use those feature vectors to create a pair of grid arrays `xx1` and `xx2` via the NumPy `meshgrid` function. Since we trained our perceptron classifier on two feature dimensions, we need to flatten the grid arrays and create a matrix that has the same number of columns as the Iris training subset so that we can use the `predict` method to predict the class labels `Z` of the corresponding grid points. After reshaping the predicted class labels `Z` into a grid with the same dimensions as `xx1` and `xx2`, we can now draw a contour plot via matplotlib's `contourf` function that maps the different decision regions to different colors for each predicted class in the grid array:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should now see a plot of the decision regions, as shown in the following figure:

As we can see in the preceding plot, the perceptron learned a decision boundary that was able to classify all flower samples in the Iris training subset perfectly.
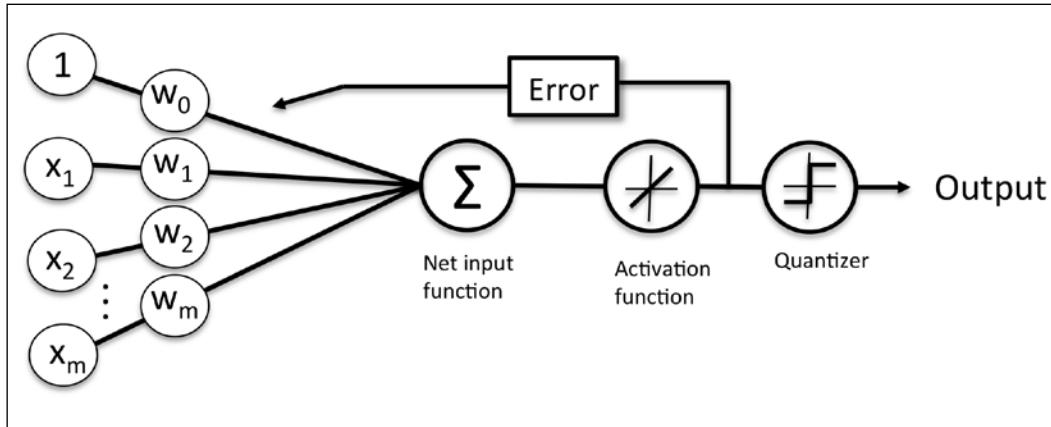
> Although the perceptron classified the two Iris flower classes perfectly, convergence is one of the biggest problems of the perceptron. Frank Rosenblatt proved mathematically that the perceptron learning rule converges if the two classes can be separated by a linear hyperplane. However, if classes cannot be separated perfectly by such a linear decision boundary, the weights will never stop updating unless we set a maximum number of epochs.

Adaptive linear neurons and the convergence of learning

In this section, we will take a look at another type of single-layer neural network: **ADAptive LInear NEuron** (**Adaline**). Adaline was published, only a few years after Frank Rosenblatt's perceptron algorithm, by Bernard Widrow and his doctoral student Tedd Hoff, and can be considered as an improvement on the latter (B. Widrow et al. Adaptive "*Adaline" neuron using chemical "memistors*". Number Technical Report 1553-2. Stanford Electron. Labs. Stanford, CA, October 1960). The Adaline algorithm is particularly interesting because it illustrates the key concept of defining and minimizing cost functions, which will lay the groundwork for understanding more advanced machine learning algorithms for classification, such as logistic regression and support vector machines, as well as regression models that we will discuss in future chapters.

The key difference between the Adaline rule (also known as the *Widrow-Hoff rule*) and Rosenblatt's perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function $\phi(z)$ is simply the identity function of the net input so that $\phi(w^T x) = w^T x$.

While the linear activation function is used for learning the weights, a *quantizer*, which is similar to the unit step function that we have seen before, can then be used to predict the class labels, as illustrated in the following figure:

If we compare the preceding figure to the illustration of the perceptron algorithm that we saw earlier, the difference is that we know to use the continuous valued output from the linear activation function to compute the model error and update the weights, rather than the binary class labels.
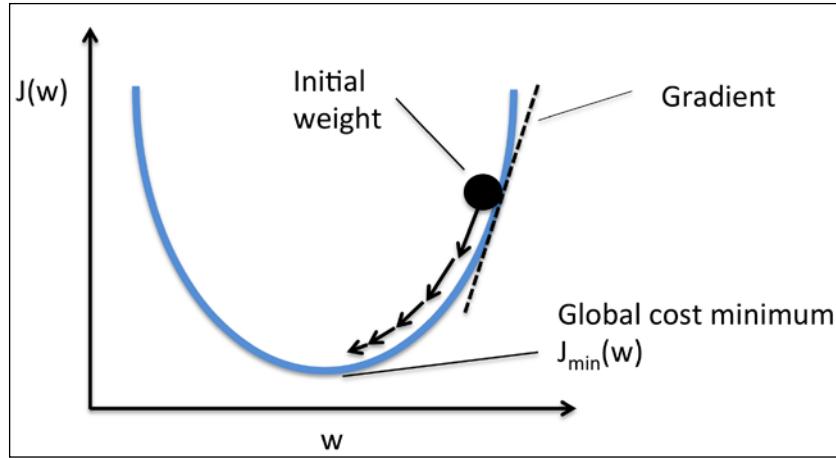
Minimizing cost functions with gradient descent

One of the key ingredients of supervised machine learning algorithms is to define an *objective function* that is to be optimized during the learning process. This objective function is often a *cost function* that we want to minimize. In the case of Adaline, we can define the cost function J to learn the weights as the **Sum of Squared Errors** (**SSE**) between the calculated outcomes and the true class labels

$$J(w) = \frac{1}{2} \sum_i \left(y^{(i)} - \phi\left(z^{(i)} \right) \right)^2 .$$

The term $\frac{1}{2}$ is just added for our convenience; it will make it easier to derive the gradient, as we will see in the following paragraphs. The main advantage of this continuous linear activation function is—in contrast to the unit step function—that the cost function becomes differentiable. Another nice property of this cost function is that it is convex; thus, we can use a simple, yet powerful, optimization algorithm called *gradient descent* to find the weights that minimize our cost function to classify the samples in the Iris dataset.

As illustrated in the following figure, we can describe the principle behind gradient descent as *climbing down a hill* until a local or global cost minimum is reached. In each iteration, we take a step away from the gradient where the step size is determined by the value of the learning rate as well as the slope of the gradient:

Using gradient descent, we can now update the weights by taking a step away from the gradient $\nabla J(w)$ of our cost function $J(w)$:

$$w := w + \Delta w$$

Here, the weight change Δw is defined as the negative gradient multiplied by the learning rate η:

$$\Delta w = -\eta \nabla J(w)$$

To compute the gradient of the cost function, we need to compute the partial derivative of the cost function with respect to each weight w_j, $\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$, so that we can write the update of weight w_j as $\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$.

Since we update all weights simultaneously, our Adaline learning rule becomes $w := w + \Delta w$.

For those who are familiar with calculus, the partial derivative of the SSE cost function with respect to the *j*th weight in can be obtained as follows:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)^2$$

$$= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)^2$$

$$= \frac{1}{2} \sum_i 2\left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)} \right) \right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)\left(-x_j^{(i)}\right)$$

$$= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

Although the Adaline learning rule looks identical to the perceptron rule, the $\phi\left(z^{(i)}\right)$ with $z^{(i)} = \boldsymbol{w}^T \boldsymbol{x}^{(i)}$ is a real number and not an integer class label. Furthermore, the weight update is calculated based on all samples in the training set (instead of updating the weights incrementally after each sample), which is why this approach is also referred to as "batch" gradient descent.

Implementing an Adaptive Linear Neuron in Python

Since the perceptron rule and Adaline are very similar, we will take the perceptron implementation that we defined earlier and change the `fit` method so that the weights are updated by minimizing the cost function via gradient descent:

```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
```

```
eta : float
    Learning rate (between 0.0 and 1.0)
n_iter : int
    Passes over the training dataset.

Attributes
-----------
w_ : 1d-array
    Weights after fitting.
errors_ : list
    Number of misclassifications in every epoch.

"""
def __init__(self, eta=0.01, n_iter=50):
    self.eta = eta
    self.n_iter = n_iter

def fit(self, X, y):
    """ Fit training data.

    Parameters
    ----------
    X : {array-like}, shape = [n_samples, n_features]
        Training vectors,
        where n_samples is the number of samples and
        n_features is the number of features.
    y : array-like, shape = [n_samples]
        Target values.

    Returns
    -------
    self : object

    """
    self.w_ = np.zeros(1 + X.shape[1])
    self.cost_ = []

    for i in range(self.n_iter):
        output = self.net_input(X)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
```

```
        cost = (errors**2).sum() / 2.0
        self.cost_.append(cost)
    return self

def net_input(self, X):
    """Calculate net input"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self, X):
    """Compute linear activation"""
    return self.net_input(X)

def predict(self, X):
    """Return class label after unit step"""
    return np.where(self.activation(X) >= 0.0, 1, -1)
```

Instead of updating the weights after evaluating each individual training sample, as in the perceptron, we calculate the gradient based on the whole training dataset via `self.eta * errors.sum()` for the zero-weight and via `self.eta * X.T.dot(errors)` for the weights 1 to *m* where `X.T.dot(errors)` is a *matrix-vector multiplication* between our feature matrix and the error vector. Similar to the previous perceptron implementation, we collect the cost values in a list `self.cost_` to check if the algorithm converged after training.

Performing a matrix-vector multiplication is similar to calculating a vector dot product where each row in the matrix is treated as a single row vector. This vectorized approach represents a more compact notation and results in a more efficient computation using NumPy. For example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

In practice, it often requires some experimentation to find a good learning rate η for optimal convergence. So, let's choose two different learning rates $\eta = 0.1$ and $\eta = 0.0001$ to start with and plot the cost functions versus the number of epochs to see how well the Adaline implementation learns from the training data.
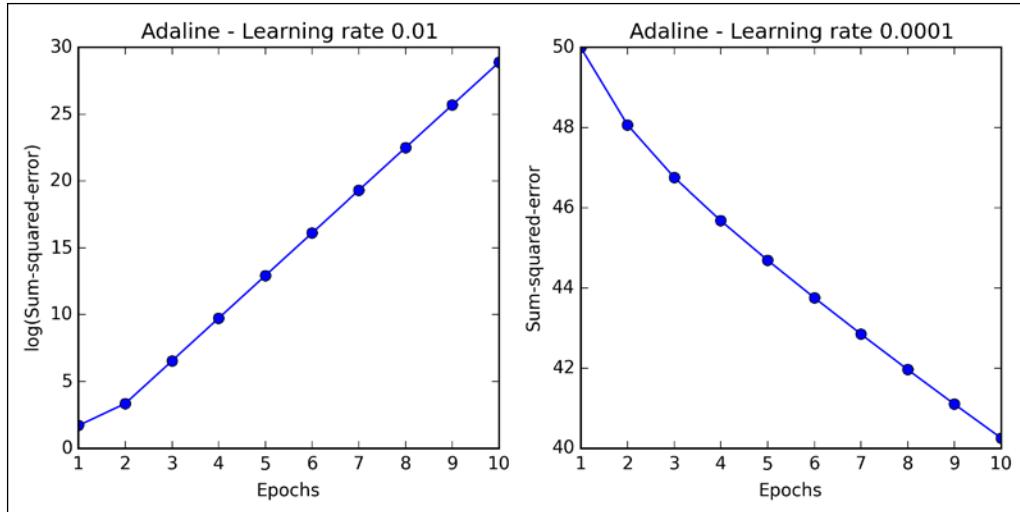
> The learning rate η, as well as the number of epochs n_iter, are the so-called *hyperparameters* of the perceptron and Adaline learning algorithms. In *Chapter 4, Building Good Training Sets – Data Preprocessing,* we will take a look at different techniques to automatically find the values of different hyperparameters that yield optimal performance of the classification model.
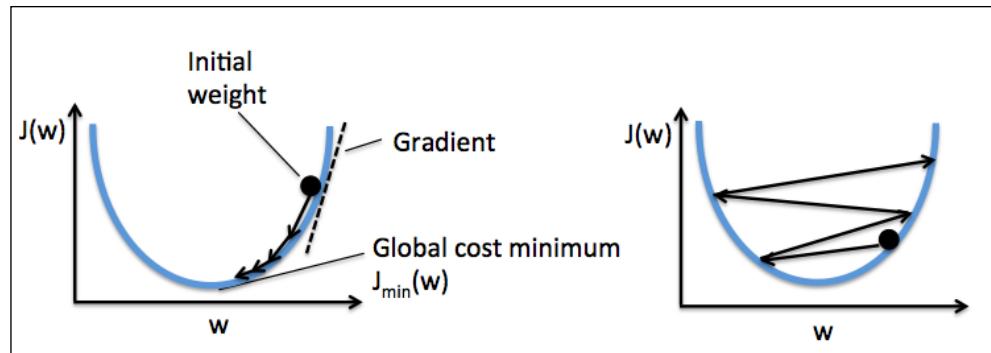
Let us now plot the cost against the number of epochs for the two different learning rates:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.cost_) + 1),
...            np.log10(ada1.cost_), marker='o')
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Sum-squared-error)')
>>> ax[0].set_title('Adaline - Learning rate 0.01')
>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.cost_) + 1),
...            ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Sum-squared-error')
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> plt.show()
```

As we can see in the resulting cost function plots next, we encountered two different types of problems. The left chart shows what could happen if we choose a learning rate that is too large—instead of minimizing the cost function, the error becomes larger in every epoch because we *overshoot* the global minimum:

Although we can see that the cost decreases when we look at the right plot, the chosen learning rate $\eta = 0.0001$ is so small that the algorithm would require a very large number of epochs to converge. The following figure illustrates how we change the value of a particular weight parameter to minimize the cost function J (left subfigure). The subfigure on the right illustrates what happens if we choose a learning rate that is too large, we overshoot the global minimum:

Many machine learning algorithms that we will encounter throughout this book require some sort of feature scaling for optimal performance, which we will discuss in more detail in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Gradient descent is one of the many algorithms that benefit from feature scaling. Here, we will use a feature scaling method called *standardization*, which gives our data the property of a standard normal distribution. The mean of each feature is centered at value 0 and the feature column has a standard deviation of 1. For example, to standardize the j th feature, we simply need to subtract the sample mean μ_j from every training sample and divide it by its standard deviation σ_j:

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Here x_j is a vector consisting of the j th feature values of all training samples n.
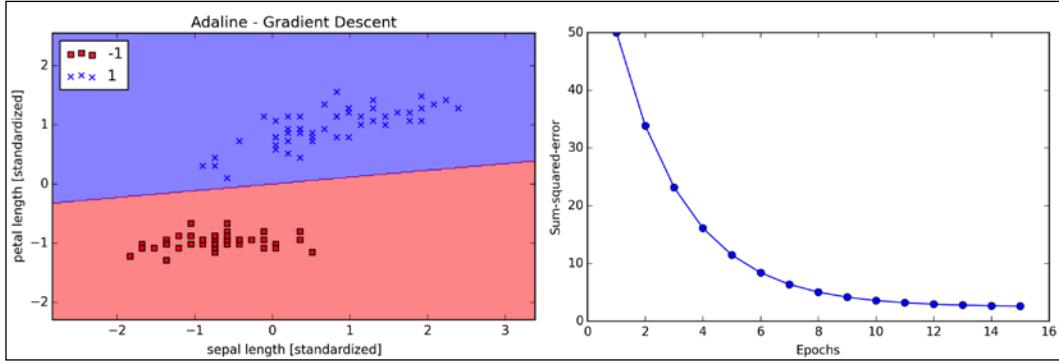
Standardization can easily be achieved using the NumPy methods `mean` and `std`:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

After standardization, we will train the Adaline again and see that it now converges using a learning rate $\eta = 0.01$:

```
>>> ada = AdalineGD(n_iter=15, eta=0.01)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Sum-squared-error')
>>> plt.show()
```

After executing the preceding code, we should see a figure of the decision regions as well as a plot of the declining cost, as shown in the following figure:

As we can see in the preceding plots, the Adaline now converges after training on the standardized features using a learning rate $\eta = 0.01$. However, note that the SSE remains non-zero even though all samples were classified correctly.

Large scale machine learning and stochastic gradient descent

In the previous section, we learned how to minimize a cost function by taking a step into the opposite direction of a gradient that is calculated from the whole training set; this is why this approach is sometimes also referred to as *batch* gradient descent. Now imagine we have a very large dataset with millions of data points, which is not uncommon in many machine learning applications. Running batch gradient descent can be computationally quite costly in such scenarios since we need to reevaluate the whole training dataset each time we take one step towards the global minimum.

A popular alternative to the batch gradient descent algorithm is *stochastic gradient descent*, sometimes also called *iterative* or *on-line* gradient descent. Instead of updating the weights based on the sum of the accumulated errors over all samples $x^{(i)}$:

$$\Delta w = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x^{(i)},$$

We update the weights incrementally for each training sample:

$$\eta \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x^{(i)}$$

Although stochastic gradient descent can be considered as an approximation of gradient descent, it typically reaches convergence much faster because of the more frequent weight updates. Since each gradient is calculated based on a single training example, the error surface is noisier than in gradient descent, which can also have the advantage that stochastic gradient descent can escape shallow local minima more readily. To obtain accurate results via stochastic gradient descent, it is important to present it with data in a random order, which is why we want to shuffle the training set for every epoch to prevent cycles.

In stochastic gradient descent implementations, the fixed learning rate η is often replaced by an adaptive learning rate that decreases over time,

for example, $\dfrac{c_1}{\left[number\ of\ iterations \right] + c_2}$ where c_1 and c_2 are constants. Note that stochastic gradient descent does not reach the global minimum but an area very close to it. By using an adaptive learning rate, we can achieve further annealing to a better global minimum

Another advantage of stochastic gradient descent is that we can use it for *online learning*. In online learning, our model is trained on-the-fly as new training data arrives. This is especially useful if we are accumulating large amounts of data—for example, customer data in typical web applications. Using online learning, the system can immediately adapt to changes and the training data can be discarded after updating the model if storage space in an issue.

A compromise between batch gradient descent and stochastic gradient descent is the so-called *mini-batch learning*. Mini-batch learning can be understood as applying batch gradient descent to smaller subsets of the training data—for example, 50 samples at a time. The advantage over batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates. Furthermore, mini-batch learning allows us to replace the for-loop over the training samples in **Stochastic Gradient Descent** (**SGD**) by vectorized operations, which can further improve the computational efficiency of our learning algorithm.

Since we already implemented the Adaline learning rule using gradient descent, we only need to make a few adjustments to modify the learning algorithm to update the weights via stochastic gradient descent. Inside the `fit` method, we will now update the weights after each training sample. Furthermore, we will implement an additional `partial_fit` method, which does not reinitialize the weights, for on-line learning. In order to check if our algorithm converged after training, we will calculate the cost as the average cost of the training samples in each epoch. Furthermore, we will add an option to `shuffle` the training data before each epoch to avoid cycles when we are optimizing the cost function; via the `random_state` parameter, we allow the specification of a random seed for consistency:

```python
from numpy.random import seed

class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.
    shuffle : bool (default: True)
        Shuffles training data every epoch
        if True to prevent cycles.
    random_state : int (default: None)
        Set random state for shuffling
        and initializing the weights.

    """
    def __init__(self, eta=0.01, n_iter=10,
                 shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
```

```
        if random_state:
            seed(random_state)

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples
            is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

         """
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost)/len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
```

```
        """Shuffle training data"""
        r = np.random.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights to zeros"""
        self.w_ = np.zeros(1 + m)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Apply Adaline learning rule to update the weights"""
        output = self.net_input(xi)
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        cost = 0.5 * error**2
        return cost

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```
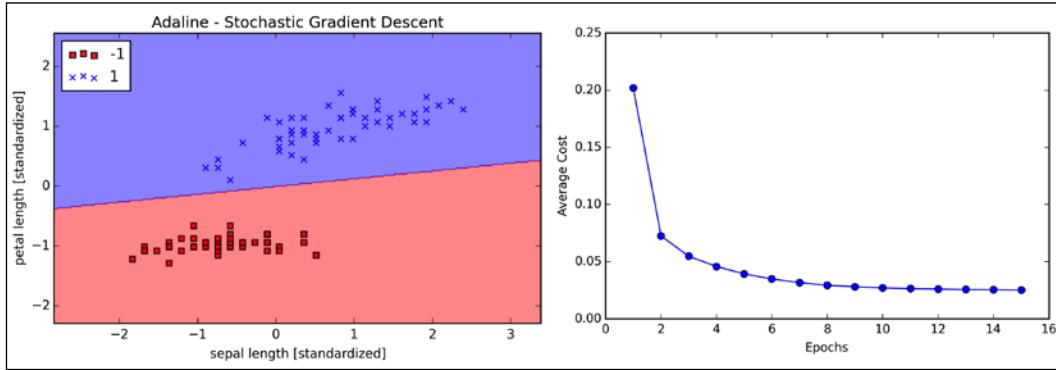
The `_shuffle` method that we are now using in the `AdalineSGD` classifier works as follows: via the `permutation` function in `numpy.random`, we generate a random sequence of unique numbers in the range 0 to 100. Those numbers can then be used as indices to shuffle our feature matrix and class label vector.

We can then use the fit method to train the `AdalineSGD` classifier and use our `plot_decision_regions` to plot our training results:

```
>>> ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Stochastic Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
```

```
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Average Cost')
>>> plt.show()
```

The two plots that we obtain from executing the preceding code example are shown in the following figure:

As we can see, the average cost goes down pretty quickly, and the final decision boundary after 15 epochs looks similar to the batch gradient descent with Adaline. If we want to update our model—for example, in an on-line learning scenario with streaming data—we could simply call the `partial_fit` method on individual samples—for instance, `ada.partial_fit(X_std[0, :], y[0])`.

Summary

In this chapter, we gained a good understanding of the basic concepts of linear classifiers for supervised learning. After we implemented a perceptron, we saw how we can train adaptive linear neurons efficiently via a vectorized implementation of gradient descent and on-line learning via stochastic gradient descent. Now that we have seen how to implement simple classifiers in Python, we are ready to move on to the next chapter where we will use the Python scikit-learn machine learning library to get access to more advanced and powerful off-the-shelf machine learning classifiers that are commonly used in academia as well as in industry.

3
A Tour of Machine Learning Classifiers Using Scikit-learn

In this chapter, we will take a tour through a selection of popular and powerful machine learning algorithms that are commonly used in academia as well as in the industry. While learning about the differences between several supervised learning algorithms for classification, we will also develop an intuitive appreciation of their individual strengths and weaknesses. Also, we will take our first steps with the scikit-learn library, which offers a user-friendly interface for using those algorithms efficiently and productively.

The topics that we will learn about throughout this chapter are as follows:

- Introduction to the concepts of popular classification algorithms
- Using the scikit-learn machine learning library
- Questions to ask when selecting a machine learning algorithm

Choosing a classification algorithm

Choosing an appropriate classification algorithm for a particular problem task requires practice: each algorithm has its own quirks and is based on certain assumptions. To restate the "No Free Lunch" theorem: no single classifier works best across all possible scenarios. In practice, it is always recommended that you compare the performance of at least a handful of different learning algorithms to select the best model for the particular problem; these may differ in the number of features or samples, the amount of noise in a dataset, and whether the classes are linearly separable or not.

Eventually, the performance of a classifier, computational power as well as predictive power, depends heavily on the underlying data that are available for learning. The five main steps that are involved in training a machine learning algorithm can be summarized as follows:

1. Selection of features.
2. Choosing a performance metric.
3. Choosing a classifier and optimization algorithm.
4. Evaluating the performance of the model.
5. Tuning the algorithm.

Since the approach of this book is to build machine learning knowledge step by step, we will mainly focus on the principal concepts of the different algorithms in this chapter and revisit topics such as feature selection and preprocessing, performance metrics, and hyperparameter tuning for more detailed discussions later in this book.

First steps with scikit-learn

In *Chapter 2*, *Training Machine Learning Algorithms for Classification*, you learned about two related learning algorithms for classification: the **perceptron** rule and **Adaline**, which we implemented in Python by ourselves. Now we will take a look at the scikit-learn API, which combines a user-friendly interface with a highly optimized implementation of several classification algorithms. However, the scikit-learn library offers not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models. We will discuss this in more detail together with the underlying concepts in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, and *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

Training a perceptron via scikit-learn

To get started with the scikit-learn library, we will train a perceptron model similar to the one that we implemented in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. For simplicity, we will use the already familiar **Iris** dataset throughout the following sections. Conveniently, the Iris dataset is already available via scikit-learn, since it is a simple yet popular dataset that is frequently used for testing and experimenting with algorithms. Also, we will only use two features from the **Iris flower** dataset for visualization purposes.

We will assign the *petal length* and *petal width* of the 150 flower samples to the feature matrix X and the corresponding class labels of the flower species to the vector y:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
```

If we executed np.unique(y) to return the different class labels stored in iris.target, we would see that the Iris flower class names, *Iris-Setosa*, *Iris-Versicolor*, and *Iris-Virginica*, are already stored as integers (0, 1, 2), which is recommended for the optimal performance of many machine learning libraries.

To evaluate how well a trained model performs on unseen data, we will further split the dataset into separate training and test datasets. Later in *Chapter 5, Compressing Data via Dimensionality Reduction*, we will discuss the best practices around model evaluation in more detail:

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...             X, y, test_size=0.3, random_state=0)
```

Using the train_test_split function from scikit-learn's cross_validation module, we randomly split the X and y arrays into 30 percent test data (45 samples) and 70 percent training data (105 samples).

Many machine learning and optimization algorithms also require feature scaling for optimal performance, as we remember from the **gradient descent** example in *Chapter 2, Training Machine Learning Algorithms for Classification*. Here, we will standardize the features using the StandardScaler class from scikit-learn's preprocessing module:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Using the preceding code, we loaded the `StandardScaler` class from the preprocessing module and initialized a new `StandardScaler` object that we assigned to the variable `sc`. Using the `fit` method, `StandardScaler` estimated the parameters μ (sample mean) and σ (standard deviation) for each feature dimension from the training data. By calling the `transform` method, we then standardized the training data using those estimated parameters μ and σ. Note that we used the same scaling parameters to standardize the test set so that both the values in the training and test dataset are comparable to each other.

Having standardized the training data, we can now train a perceptron model. Most algorithms in scikit-learn already support multiclass classification by default via the **One-vs.-Rest (OvR)** method, which allows us to feed the three flower classes to the perceptron all at once. The code is as follows:

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)
>>> ppn.fit(X_train_std, y_train)
```

The scikit-learn interface reminds us of our perceptron implementation in *Chapter 2, Training Machine Learning Algorithms for Classification*: after loading the `Perceptron` class from the `linear_model` module, we initialized a new `Perceptron` object and trained the model via the `fit` method. Here, the model parameter `eta0` is equivalent to the learning rate `eta` that we used in our own perceptron implementation, and the parameter `n_iter` defines the number of epochs (passes over the training set). As we remember from *Chapter 2, Training Machine Learning Algorithms for Classification*, finding an appropriate learning rate requires some experimentation. If the learning rate is too large, the algorithm will overshoot the global cost minimum. If the learning rate is too small, the algorithm requires more epochs until convergence, which can make the learning slow—especially for large datasets. Also, we used the `random_state` parameter for reproducibility of the initial shuffling of the training dataset after each epoch.

Having trained a model in scikit-learn, we can make predictions via the `predict` method, just like in our own perceptron implementation in *Chapter 2, Training Machine Learning Algorithms for Classification*. The code is as follows:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified samples: %d' % (y_test != y_pred).sum())
Misclassified samples: 4
```

On executing the preceding code, we see that the perceptron misclassifies 4 out of the 45 flower samples. Thus, the misclassification error on the test dataset is 0.089 or 8.9 percent ($4/45 \approx 0.089$).

> Instead of the misclassification **error**, many machine learning practitioners report the classification **accuracy** of a model, which is simply calculated as follows:
>
> 1 - *misclassification error* = 0.911 or 91.1 percent.

Scikit-learn also implements a large variety of different performance metrics that are available via the `metrics` module. For example, we can calculate the classification accuracy of the perceptron on the test set as follows:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
0.91
```

Here, `y_test` are the true class labels and `y_pred` are the class labels that we predicted previously.

> Note that we evaluate the performance of our models based on the test set in this chapter. In *Chapter 5*, *Compressing Data via Dimensionality Reduction*, you will learn about useful techniques, including graphical analysis such as learning curves, to detect and prevent **overfitting**. Overfitting means that the model captures the patterns in the training data well, but fails to generalize well to unseen data.

Finally, we can use our `plot_decision_regions` function from *Chapter 2*, *Training Machine Learning Algorithms for Classification*, to plot the **decision regions** of our newly trained perceptron model and visualize how well it separates the different flower samples. However, let's add a small modification to highlight the samples from the test dataset via small circles:

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def plot_decision_regions(X, y, classifier,
                    test_idx=None, resolution=0.02):

    # setup marker generator and color map
```

```
        markers = ('s', 'x', 'o', '^', 'v')
        colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
        cmap = ListedColormap(colors[:len(np.unique(y))])

        # plot the decision surface
        x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                               np.arange(x2_min, x2_max, resolution))
        Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
        Z = Z.reshape(xx1.shape)
        plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
        plt.xlim(xx1.min(), xx1.max())
        plt.ylim(xx2.min(), xx2.max())

        # plot all samples
        for idx, cl in enumerate(np.unique(y)):
            plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                        alpha=0.8, c=cmap(idx),
                        marker=markers[idx], label=cl)

        # highlight test samples
        if test_idx:
            X_test, y_test = X[test_idx, :], y[test_idx]
            plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                        alpha=1.0, linewidths=1, marker='o',
                        s=55, label='test set')
```
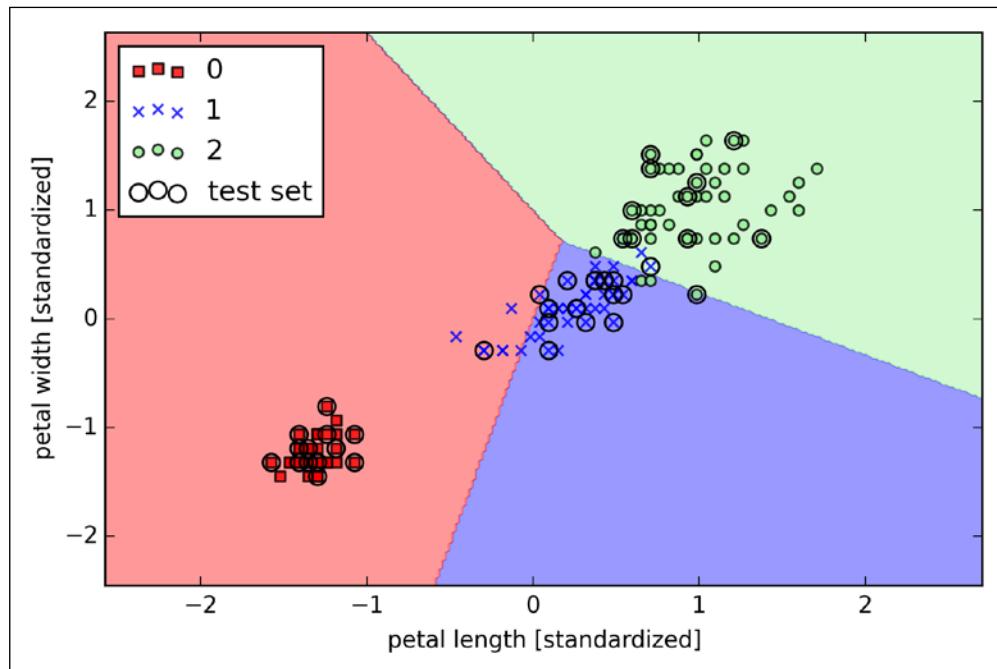
With the slight modification that we made to the plot_decision_regions function (highlighted in the preceding code), we can now specify the indices of the samples that we want to mark on the resulting plots. The code is as follows:

```
>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                       y=y_combined,
...                       classifier=ppn,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

As we can see in the resulting plot, the three flower classes cannot be perfectly separated by a linear decision boundaries:

We remember from our discussion in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, that the perceptron algorithm never converges on datasets that aren't perfectly linearly separable, which is why the use of the perceptron algorithm is typically not recommended in practice. In the following sections, we will look at more powerful linear classifiers that converge to a cost minimum even if the classes are not perfectly linearly separable.

> The `Perceptron` as well as other scikit-learn functions and classes have additional parameters that we omit for clarity. You can read more about those parameters using the `help` function in Python (for example, `help(Perceptron)`) or by going through the excellent scikit-learn online documentation at `http://scikit-learn.org/stable/`.

Modeling class probabilities via logistic regression

Although the perceptron rule offers a nice and easygoing introduction to machine learning algorithms for classification, its biggest disadvantage is that it never converges if the classes are not perfectly linearly separable. The classification task in the previous section would be an example of such a scenario. Intuitively, we can think of the reason as the weights are continuously being updated since there is always at least one misclassified sample present in each epoch. Of course, you can change the learning rate and increase the number of epochs, but be warned that the perceptron will never converge on this dataset. To make better use of our time, we will now take a look at another simple yet more powerful algorithm for linear and binary classification problems: **logistic regression**. Note that, in spite of its name, logistic regression is a model for classification, not regression.

Logistic regression intuition and conditional probabilities

Logistic regression is a classification model that is very easy to implement but performs very well on linearly separable classes. It is one of the most widely used algorithms for classification in industry. Similar to the perceptron and Adaline, the logistic regression model in this chapter is also a linear model for binary classification that can be extended to multiclass classification via the OvR technique.

To explain the idea behind logistic regression as a probabilistic model, let's first introduce the **odds ratio**, which is the odds in favor of a particular event. The odds ratio can be written as $\frac{p}{(1-p)}$, where p stands for the probability of the positive event. The term *positive event* does not necessarily mean *good*, but refers to the event that we want to predict, for example, the probability that a patient has a certain disease; we can think of the positive event as class label $y = 1$. We can then further define the **logit** function, which is simply the logarithm of the odds ratio (log-odds):

$$logit(p) = log\frac{p}{(1-p)}$$

The logit function takes input values in the range 0 to 1 and transforms them to values over the entire real number range, which we can use to express a linear relationship between feature values and the log-odds:

$$logit\left(p\left(y=1\mid \boldsymbol{x}\right)\right)=w_0x_0+w_1x_1+\cdots+w_mx_m=\sum_{i=0}^{m}w_ix_i=\boldsymbol{w}^T\boldsymbol{x}$$

Here, $p(y=1\mid \boldsymbol{x})$ is the conditional probability that a particular sample belongs to class 1 given its features x.

Now what we are actually interested in is predicting the probability that a certain sample belongs to a particular class, which is the inverse form of the logit function. It is also called the *logistic* function, sometimes simply abbreviated as *sigmoid* function due to its characteristic S-shape.
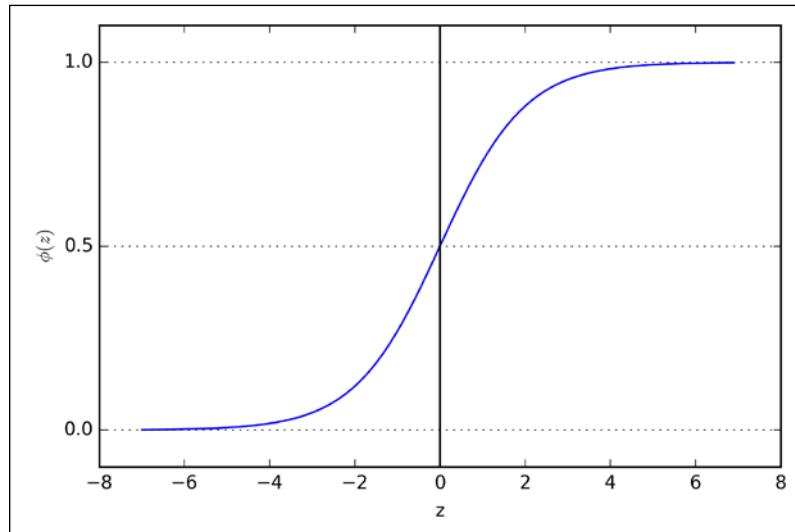
$$\phi(z)=\frac{1}{1+e^{-z}}$$

Here, z is the net input, that is, the linear combination of weights and sample features and can be calculated as $z=\boldsymbol{w}^T\boldsymbol{x}=w_0+w_1x_1+\cdots+w_mx_m$.

Now let's simply plot the sigmoid function for some values in the range -7 to 7 to see what it looks like:
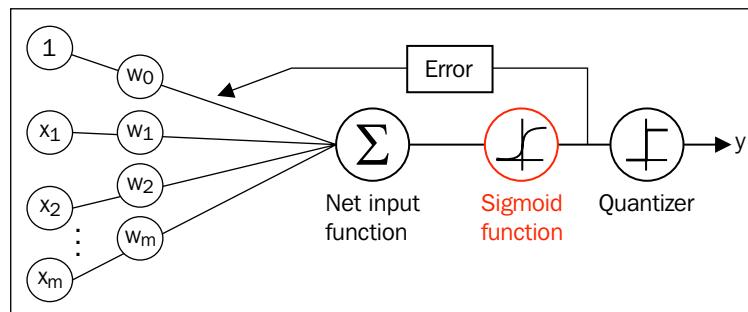
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.axhspan(0.0, 1.0, facecolor='1.0', alpha=1.0, ls='dotted')
>>> plt.axhline(y=0.5, ls='dotted', color='k')
>>> plt.yticks([0.0, 0.5, 1.0])
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi (z)$')
>>> plt.show()
```

As a result of executing the previous code example, we should now see the **S-shaped** (sigmoidal) curve:

We can see that $\phi(z)$ approaches 1 if z goes towards infinity ($z \to \infty$), since e^{-z} becomes very small for large values of z. Similarly, $\phi(z)$ goes towards 0 for $z \to -\infty$ as the result of an increasingly large denominator. Thus, we conclude that this sigmoid function takes real number values as input and transforms them to values in the range [0, 1] with an intercept at $\phi(z) = 0.5$.

To build some intuition for the logistic regression model, we can relate it to our previous Adaline implementation in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. In Adaline, we used the identity function $\phi(z) = z$ as the activation function. In logistic regression, this activation function simply becomes the sigmoid function that we defined earlier, which is illustrated in the following figure:

The output of the sigmoid function is then interpreted as the probability of particular sample belonging to class 1 $\phi(z) = P(y = 1 \mid x; w)$, given its features x parameterized by the weights w. For example, if we compute $\phi(z) = 0.8$ for a particular flower sample, it means that the chance that this sample is an Iris-Versicolor flower is 80 percent. Similarly, the probability that this flower is an Iris-Setosa flower can be calculated as $P(y = 0 \mid x; w) = 1 - P(y = 1 \mid x; w) = 0.2$ or 20 percent. The predicted probability can then simply be converted into a binary outcome via a quantizer (unit step function):

$$\hat{y} = \begin{cases} 1 & if\,\phi(z) \geq 0.5 \\ 0 & otherwise \end{cases}$$

If we look at the preceding sigmoid plot, this is equivalent to the following:

$$\hat{y} = \begin{cases} 1 & if\,z \geq 0.0 \\ 0 & otherwise \end{cases}$$

In fact, there are many applications where we are not only interested in the predicted class labels, but where estimating the class-membership probability is particularly useful. Logistic regression is used in weather forecasting, for example, to not only predict if it will rain on a particular day but also to report the chance of rain. Similarly, logistic regression can be used to predict the chance that a patient has a particular disease given certain symptoms, which is why logistic regression enjoys wide popularity in the field of medicine.

Learning the weights of the logistic cost function

You learned how we could use the logistic regression model to predict probabilities and class labels. Now let's briefly talk about the parameters of the model, for example, weights w. In the previous chapter, we defined the sum-squared-error cost function:

$$J(w) = \sum_i \frac{1}{2} \left(\phi\left(z^{(i)}\right) - y^{(i)} \right)^2$$

We minimized this in order to learn the weights w for our Adaline classification model. To explain how we can derive the cost function for logistic regression, let's first define the likelihood L that we want to maximize when we build a logistic regression model, assuming that the individual samples in our dataset are independent of one another. The formula is as follows:

$$L(w) = P(y \mid x; w) = \prod_{i=1}^{n} P\left(y^{(i)} \mid x^{(i)}; w\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

In practice, it is easier to maximize the (natural) log of this equation, which is called the log-likelihood function:

$$l(w) = \log L(w) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$

Firstly, applying the log function reduces the potential for numerical underflow, which can occur if the likelihoods are very small. Secondly, we can convert the product of factors into a summation of factors, which makes it easier to obtain the derivative of this function via the addition trick, as you may remember from calculus.

Now we could use an optimization algorithm such as gradient ascent to maximize this log-likelihood function. Alternatively, let's rewrite the log-likelihood as a cost function J that can be minimized using gradient descent as in *Chapter 2*, *Training Machine Learning Algorithms for Classification*:

$$J(w) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$
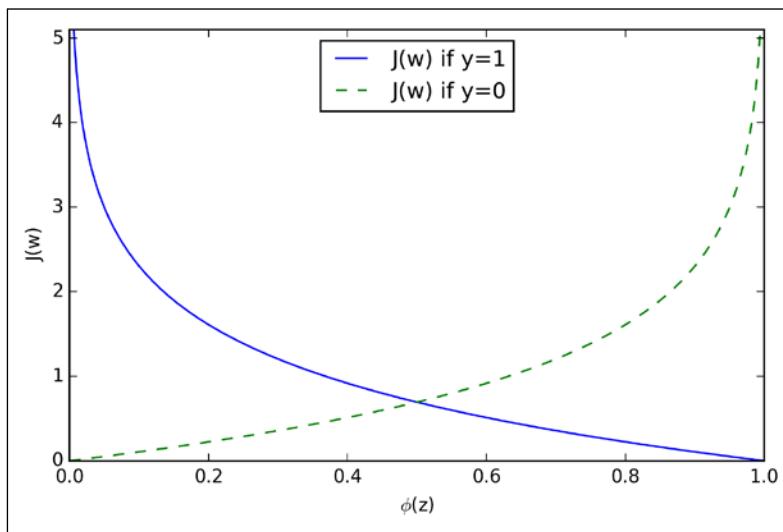
To get a better grasp on this cost function, let's take a look at the cost that we calculate for one single-sample instance:

$$J\left(\phi(z), y; w\right) = -y \log\left(\phi(z)\right) - (1 - y) \log\left(1 - \phi(z)\right)$$

Looking at the preceding equation, we can see that the first term becomes zero if $y = 0$, and the second term becomes zero if $y = 1$, respectively:

$$J\big(\phi(z), y; \boldsymbol{w}\big) = \begin{cases} -\log\big(\phi(z)\big) & \textit{if } y = 1 \\ -\log\big(1 - \phi(z)\big) & \textit{if } y = 0 \end{cases}$$

The following plot illustrates the cost for the classification of a single-sample instance for different values of $\phi(z)$:

We can see that the cost approaches 0 (plain blue line) if we correctly predict that a sample belongs to class 1. Similarly, we can see on the *y* axis that the cost also approaches 0 if we correctly predict $y = 0$ (dashed line). However, if the prediction is wrong, the cost goes towards infinity. The moral is that we penalize wrong predictions with an increasingly larger cost.

Training a logistic regression model with scikit-learn
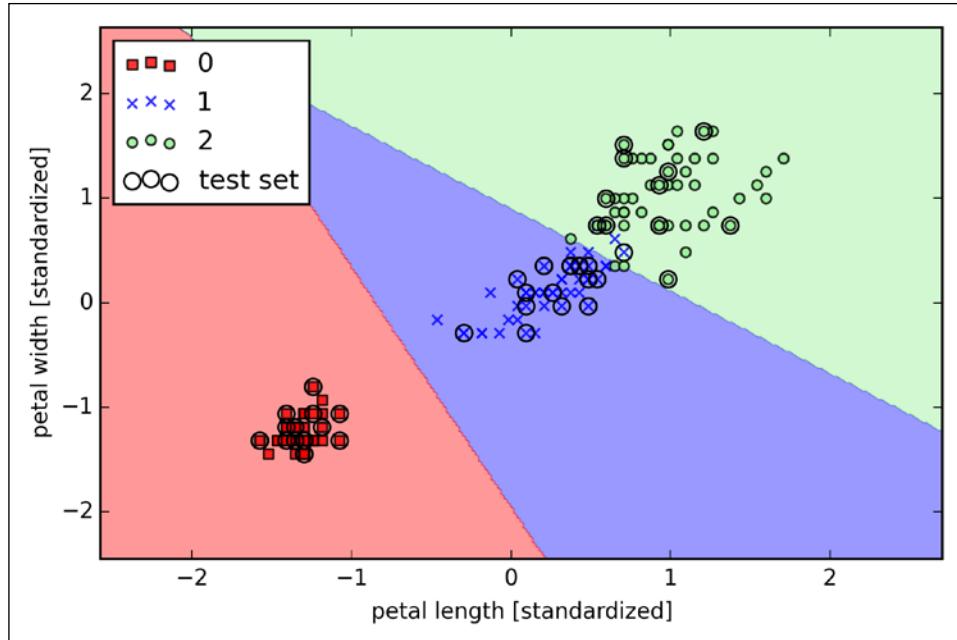
If we were to implement logistic regression ourselves, we could simply substitute the cost function J in our Adaline implementation from *Chapter 2*, *Training Machine Learning Algorithms for Classification*, by the new cost function:

$$J(w) = -\sum_i y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)$$

This would compute the cost of classifying all training samples per epoch and we would end up with a working logistic regression model. However, since scikit-learn implements a highly optimized version of logistic regression that also supports multiclass settings off-the-shelf, we will skip the implementation and use the `sklearn.linear_model.LogisticRegression` class as well as the familiar `fit` method to train the model on the standardized flower training dataset:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=1000.0, random_state=0)
>>> lr.fit(X_train_std, y_train)
  >>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=lr,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After fitting the model on the training data, we plotted the decision regions, training samples and test samples, as shown here:

Looking at the preceding code that we used to train the `LogisticRegression` model, you might now be wondering, "What is this mysterious parameter `C`?" We will get to this in a second, but let's briefly go over the concept of overfitting and regularization in the next subsection first.

Furthermore, we can predict the class-membership probability of the samples via the `predict_proba` method. For example, we can predict the probabilities of the first Iris-Setosa sample:

```
>>> lr.predict_proba(X_test_std[0,:])
```

This returns the following array:

```
array([[ 0.000,   0.063,   0.937]])
```

The preceding array tells us that the model predicts a chance of 93.7 percent that the sample belongs to the Iris-Virginica class, and a 6.3 percent chance that the sample is a Iris-Versicolor flower.

We can show that the weight update in logistic regression via gradient descent is indeed equal to the equation that we used in Adaline in *Chapter 2, Training Machine Learning Algorithms for Classification*. Let's start by calculating the partial derivative of the log-likelihood function with respect to the *j*th weight:

$$\frac{\partial}{\partial w_j} l(\boldsymbol{w}) = \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

Before we continue, let's calculate the partial derivative of the sigmoid function first:

$$\frac{\partial}{\partial z} \phi(z) = \frac{\partial}{\partial z} \frac{1}{1+e^{-z}} = \frac{1}{\left(1+e^{-z}\right)^2} e^{-z} = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}} \right)$$

$$= \phi(z) \left(1 - \phi(z) \right)$$

Now we can resubstitute $\frac{\partial}{\partial z} \phi(z) = \phi(z)(1-\phi(z))$ in our first equation to obtain the following:

$$\left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

$$= \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \phi(z)(1-\phi(z)) \frac{\partial}{\partial w_j} z$$

$$= \left(y(1-\phi(z)) - (1-y)\phi(z) \right) x_j$$

$$= \left(y - \phi(z) \right) x_j$$

Remember that the goal is to find the weights that maximize the log-likelihood so that we would perform the update for each weight as follows:

$$w_j := w_j + \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x_j^{(i)}$$

Since we update all weights simultaneously, we can write the general update rule as follows:

$$w := w + \Delta w$$

We define Δw as follows:

$$\Delta w = \eta \nabla l(w)$$

Since maximizing the log-likelihood is equal to minimizing the cost function J that we defined earlier, we can write the gradient descent update rule as follows:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x_j^{(i)}$$
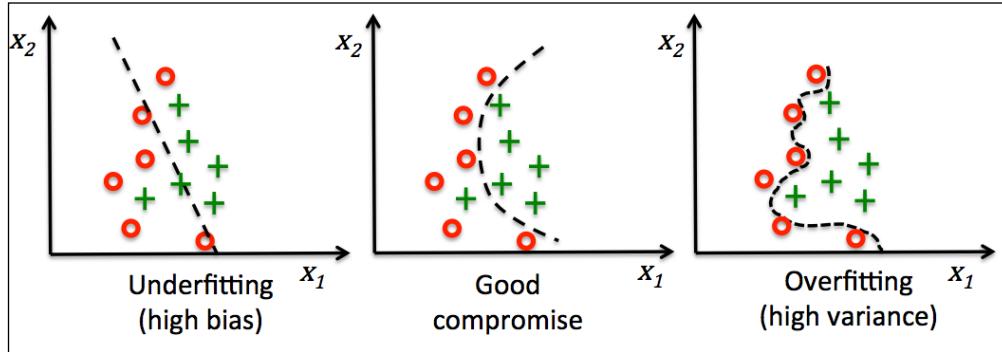
$$w := w + \Delta w, \; \Delta w = -\eta \nabla J(w)$$

This is equal to the gradient descent rule in Adaline in *Chapter 2*, *Training Machine Learning Algorithms for Classification*.

Tackling overfitting via regularization

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from overfitting, we also say that the model has a high variance, which can be caused by having too many parameters that lead to a model that is too complex given the underlying data. Similarly, our model can also suffer from **underfitting** (high bias), which means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data.

Although we have only encountered linear models for classification so far, the problem of overfitting and underfitting can be best illustrated by using a more complex, nonlinear decision boundary as shown in the following figure:
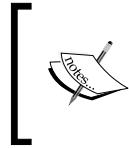
Variance measures the consistency (or variability) of the model prediction for a particular sample instance if we would retrain the model multiple times, for example, on different subsets of the training dataset. We can say that the model is sensitive to the randomness in the training data. In contrast, bias measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets; bias is the measure of the systematic error that is not due to randomness.

One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization. Regularization is a very useful method to handle collinearity (high correlation among features), filter out noise from data, and eventually prevent overfitting. The concept behind regularization is to introduce additional information (bias) to penalize extreme parameter weights. The most common form of regularization is the so-called **L2 regularization** (sometimes also called L2 shrinkage or weight decay), which can be written as follows:

$$\frac{\lambda}{2}\|w\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

Here, λ is the so-called regularization parameter.

> Regularization is another reason why feature scaling such as standardization is important. For regularization to work properly, we need to ensure that all our features are on comparable scales.

In order to apply regularization, we just need to add the regularization term to the cost function that we defined for logistic regression to shrink the weights:

$$J(\mathbf{w}) = \sum_{i=1}^{n}\left[-y^{(i)}\log\left(\phi\left(z^{(i)}\right)\right)-\left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)\right]+\frac{\lambda}{2}\|w\|^2$$

Via the regularization parameter λ, we can then control how well we fit the training data while keeping the weights small. By increasing the value of λ, we increase the regularization strength.

The parameter C that is implemented for the `LogisticRegression` class in scikit-learn comes from a convention in support vector machines, which will be the topic of the next section. C is directly related to the regularization parameter λ, which is its inverse:

$$C = \frac{1}{\lambda}$$

So we can rewrite the regularized cost function of logistic regression as follows:
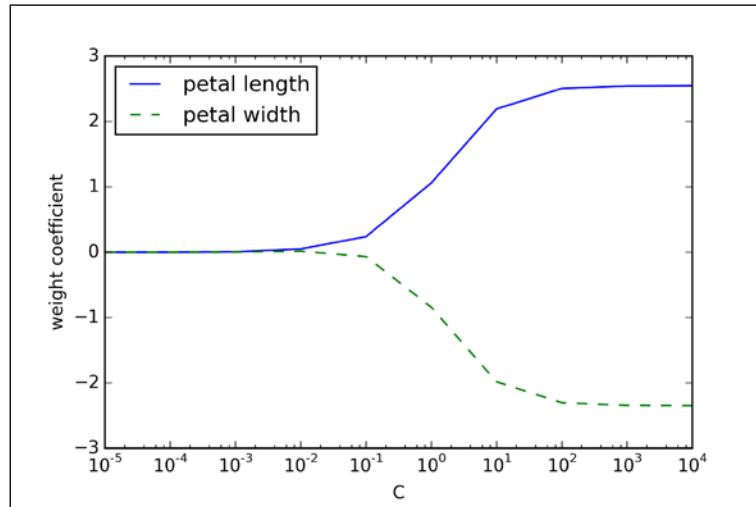
$$J(\mathbf{w}) = C\left[\sum_{i=1}^{n}\left(-y^{(i)}\log\left(\phi\left(z^{(i)}\right)\right)-\left(1-y^{(i)}\right)\right)\log\left(1-\phi\left(z^{(i)}\right)\right)\right]+\frac{1}{2}\|\mathbf{w}\|^2$$

Consequently, decreasing the value of the inverse regularization parameter C means that we are increasing the regularization strength, which we can visualize by plotting the L2 regularization path for the two weight coefficients:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10**c, random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...          label='petal length')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...          label='petal width')
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

By executing the preceding code, we fitted ten logistic regression models with different values for the inverse-regularization parameter C. For the purposes of illustration, we only collected the weight coefficients of the class 2 vs. all classifier. Remember that we are using the OvR technique for multiclass classification.
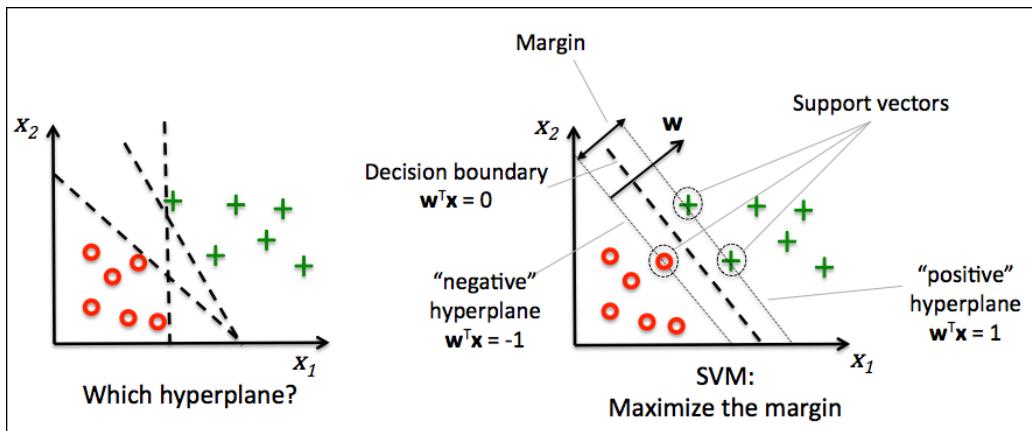
As we can see in the resulting plot, the weight coefficients shrink if we decrease the parameter **C**, that is, if we increase the regularization strength:

Since an in-depth coverage of the individual classification algorithms exceeds the scope of this book, I warmly recommend Dr. Scott Menard's *Logistic Regression: From Introductory to Advanced Concepts and Applications*, *Sage Publications*, to readers who want to learn more about logistic regression.

Maximum margin classification with support vector machines

Another powerful and widely used learning algorithm is the **support vector machine** (**SVM**), which can be considered as an extension of the perceptron. Using the perceptron algorithm, we minimized misclassification errors. However, in SVMs, our optimization objective is to maximize the **margin**. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training samples that are closest to this hyperplane, which are the so-called **support vectors**. This is illustrated in the following figure:

Maximum margin intuition

The rationale behind having decision boundaries with large margins is that they tend to have a lower generalization error whereas models with small margins are more prone to overfitting. To get an intuition for the margin maximization, let's take a closer look at those *positive* and *negative* hyperplanes that are parallel to the decision boundary, which can be expressed as follows:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{pos} = 1 \quad (1)$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{neg} = -1 \quad (2)$$

If we subtract those two linear equations (**1**) and (**2**) from each other, we get:

$$\Rightarrow \boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg} \right) = 2$$

We can normalize this by the length of the vector w, which is defined as follows:

$$\|\boldsymbol{w}\| = \sqrt{\sum_{j=1}^{m} w_j^2}$$

So we arrive at the following equation:

$$\frac{\boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg} \right)}{\|\boldsymbol{w}\|} = \frac{2}{\|\boldsymbol{w}\|}$$

The left side of the preceding equation can then be interpreted as the distance between the positive and negative hyperplane, which is the so-called margin that we want to maximize.

Now the objective function of the SVM becomes the maximization of this margin by maximizing $\frac{2}{\|w\|}$ under the constraint that the samples are classified correctly, which can be written as follows:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 \; if \; y^{(i)} = 1$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} < -1 \; if \; y^{(i)} = -1$$

These two equations basically say that all negative samples should fall on one side of the negative hyperplane, whereas all the positive samples should fall behind the positive hyperplane. This can also be written more compactly as follows:

$$y^{(i)} \left(w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \right) \geq 1 \; \forall_i$$

In practice, though, it is easier to minimize the reciprocal term $\frac{1}{2}\|\boldsymbol{w}\|^2$, which can be solved by quadratic programming. However, a detailed discussion about quadratic programming is beyond the scope of this book, but if you are interested, you can learn more about **Support Vector Machines** (**SVM**) in Vladimir Vapnik's *The Nature of Statistical Learning Theory, Springer Science & Business Media*, or Chris J.C. Burges' excellent explanation in *A Tutorial on Support Vector Machines for Pattern Recognition* (Data mining and knowledge discovery, 2(2):121–167, 1998).

Dealing with the nonlinearly separable case using slack variables

Although we don't want to dive much deeper into the more involved mathematical concepts behind the margin classification, let's briefly mention the slack variable ξ. It was introduced by Vladimir Vapnik in 1995 and led to the so-called soft-margin classification. The motivation for introducing the slack variable ξ was that the linear constraints need to be relaxed for nonlinearly separable data to allow convergence of the optimization in the presence of misclassifications under the appropriate cost penalization.

The positive-values slack variable is simply added to the linear constraints:
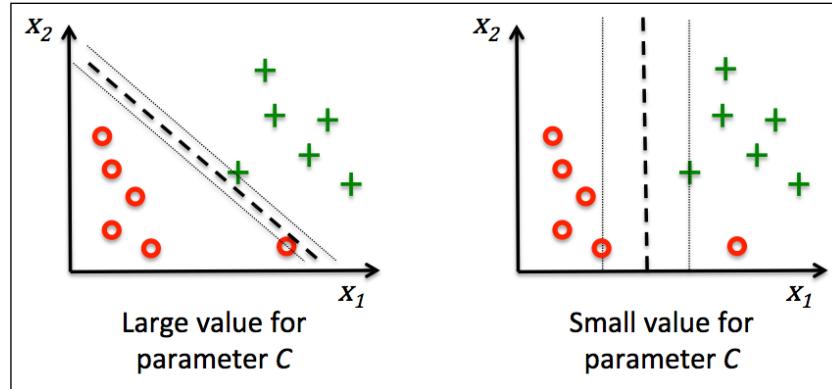
$$w^T x^{(i)} \geq 1 - \xi^{(i)} \; \text{if} \; y^{(i)} = 1$$

$$w^T x^{(i)} \leq -1 + \xi^{(i)} \; \text{if} \; y^{(i)} = -1$$

So the new objective to be minimized (subject to the preceding constraints) becomes:

$$\frac{1}{2}\|w\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

Using the variable C, we can then control the penalty for misclassification. Large values of C correspond to large error penalties whereas we are less strict about misclassification errors if we choose smaller values for C. We can then we use the parameter C to control the width of the margin and therefore tune the bias-variance trade-off as illustrated in the following figure:
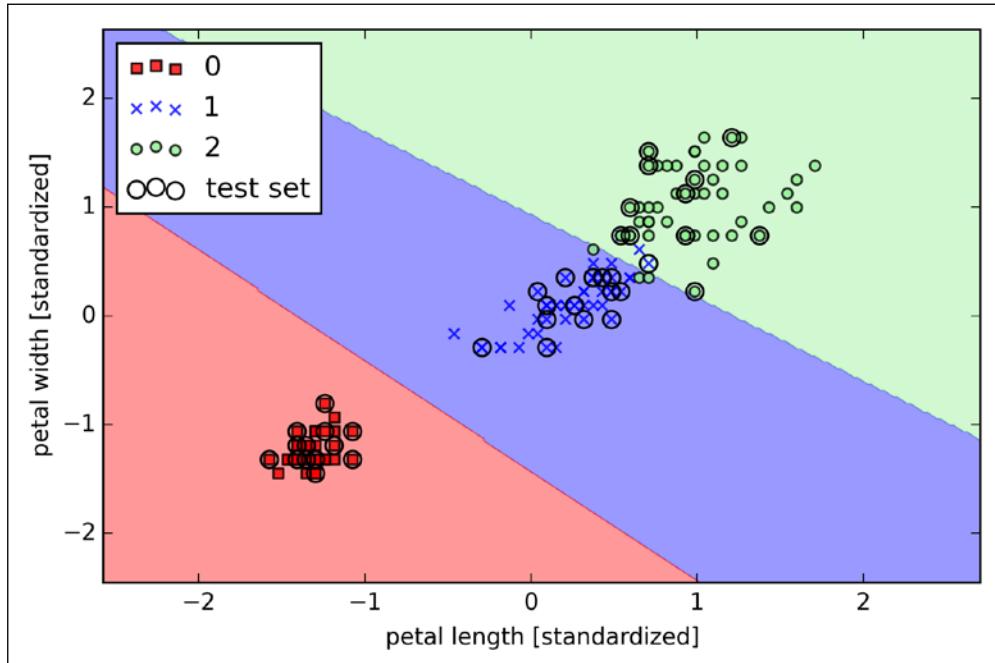
This concept is related to regularization, which we discussed previously in the context of regularized regression where increasing the value of C increases the bias and lowers the variance of the model.

Now that we learned the basic concepts behind the linear SVM, let's train a SVM model to classify the different flowers in our Iris dataset:

```
>>> from sklearn.svm import SVC
   >>> svm = SVC(kernel='linear', C=1.0, random_state=0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

The decision regions of the SVM visualized after executing the preceding code example are shown in the following plot:

Logistic regression versus SVM

In practical classification tasks, linear logistic regression and linear SVMs often yield very similar results. Logistic regression tries to maximize the conditional likelihoods of the training data, which makes it more prone to outliers than SVMs. The SVMs mostly care about the points that are closest to the decision boundary (support vectors). On the other hand, logistic regression has the advantage that it is a simpler model that can be implemented more easily. Furthermore, logistic regression models can be easily updated, which is attractive when working with streaming data.

Alternative implementations in scikit-learn

The `Perceptron` and `LogisticRegression` classes that we used in the previous sections via scikit-learn make use of the LIBLINEAR library, which is a highly optimized C/C++ library developed at the National Taiwan University (`http://www.csie.ntu.edu.tw/~cjlin/liblinear/`). Similarly, the `SVC` class that we used to train an SVM makes use of LIBSVM, which is an equivalent C/C++ library specialized for SVMs (`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`).

The advantage of using LIBLINEAR and LIBSVM over native Python implementations is that they allow an extremely quick training of large amounts of linear classifiers. However, sometimes our datasets are too large to fit into computer memory. Thus, scikit-learn also offers alternative implementations via the `SGDClassifier` class, which also supports online learning via the `partial_fit` method. The concept behind the `SGDClassifier` class is similar to the stochastic gradient algorithm that we implemented in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, for Adaline. We could initialize the stochastic gradient descent version of the perceptron, logistic regression, and support vector machine with default parameters as follows:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

Solving nonlinear problems using a kernel SVM
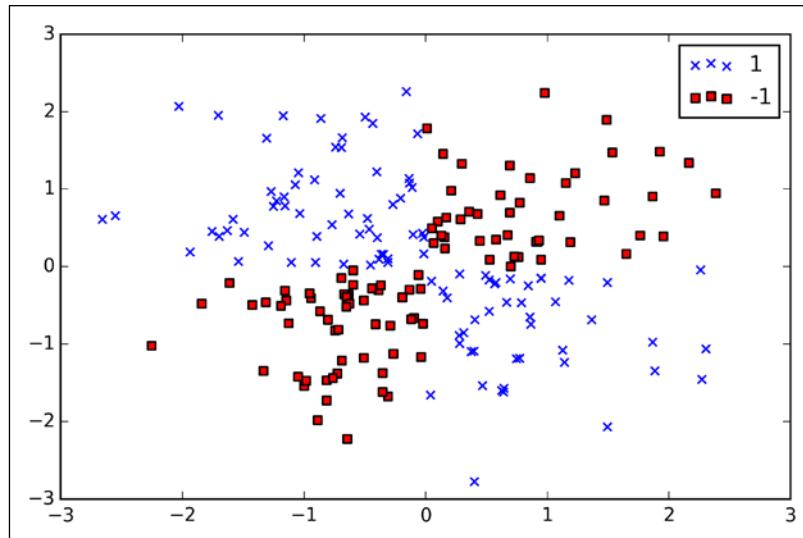
Another reason why SVMs enjoy high popularity among machine learning practitioners is that they can be easily *kernelized* to solve nonlinear classification problems. Before we discuss the main concept behind **kernel SVM**, let's first define and create a sample dataset to see how such a nonlinear classification problem may look.

Using the following code, we will create a simple dataset that has the form of an XOR gate using the `logical_xor` function from NumPy, where 100 samples will be assigned the class label 1 and 100 samples will be assigned the class label -1, respectively:

```
>>> np.random.seed(0)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0, X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, -1)

>>> plt.scatter(X_xor[y_xor==1, 0], X_xor[y_xor==1, 1],
...             c='b', marker='x', label='1')
>>> plt.scatter(X_xor[y_xor==-1, 0], X_xor[y_xor==-1, 1],
...             c='r', marker='s', label='-1')
>>> plt.ylim(-3.0)
>>> plt.legend()
>>> plt.show()
```

After executing the code, we will have an XOR dataset with random noise, as shown in the following figure:
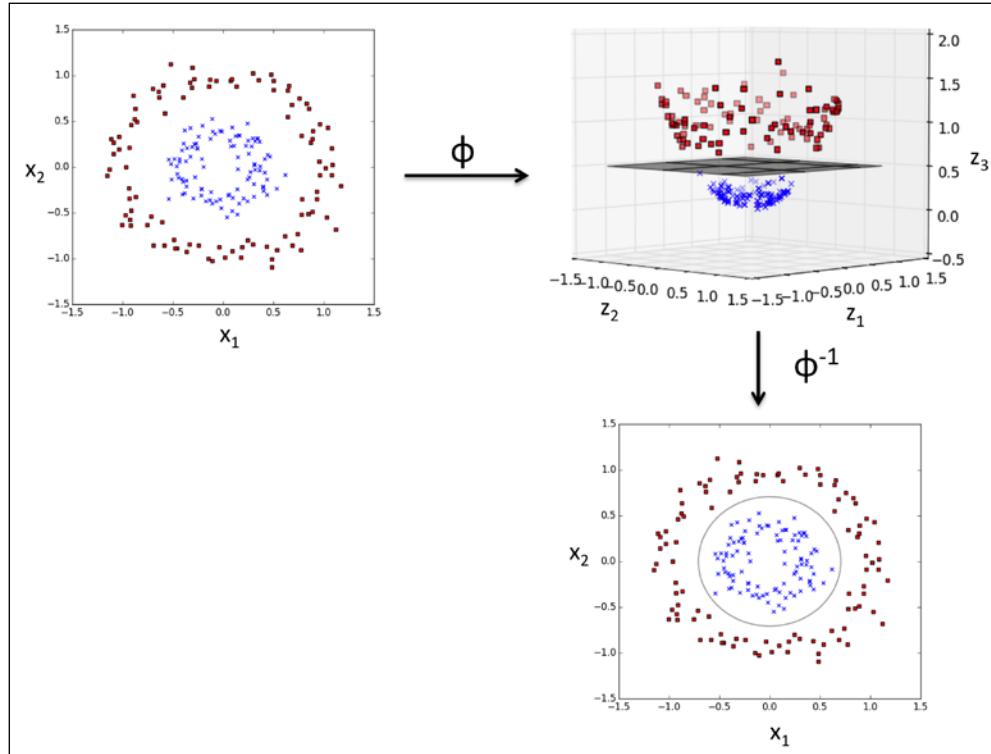
Obviously, we would not be able to separate samples from the positive and negative class very well using a linear hyperplane as the decision boundary via the linear logistic regression or linear SVM model that we discussed in earlier sections.

The basic idea behind kernel methods to deal with such linearly inseparable data is to create nonlinear combinations of the original features to project them onto a higher dimensional space via a mapping function $\phi(\cdot)$ where it becomes linearly separable. As shown in the next figure, we can transform a two-dimensional dataset onto a new three-dimensional feature space where the classes become separable via the following projection:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = \left(x_1, x_2, x_1^2 + x_2^2\right)$$

This allows us to separate the two classes shown in the plot via a linear hyperplane that becomes a nonlinear decision boundary if we project it back onto the original feature space:

Using the kernel trick to find separating hyperplanes in higher dimensional space

To solve a nonlinear problem using an SVM, we transform the training data onto a higher dimensional feature space via a mapping function $\phi(\cdot)$ and train a linear SVM model to classify the data in this new feature space. Then we can use the same mapping function $\phi(\cdot)$ to transform new, unseen data to classify it using the linear SVM model.

However, one problem with this mapping approach is that the construction of the new features is computationally very expensive, especially if we are dealing with high-dimensional data. This is where the so-called kernel trick comes into play. Although we didn't go into much detail about how to solve the quadratic programming task to train an SVM, in practice all we need is to replace the dot product $x^{(i)T} x^{(j)}$ by $\phi\left(x^{(i)}\right)^T \phi\left(x^{(j)}\right)$. In order to save the expensive step of calculating this dot product between two points explicitly, we define a so-called kernel function: $k\left(x^{(i)}, x^{(j)}\right) = \phi\left(x^{(i)}\right)^T \phi\left(x^{(j)}\right)$.

One of the most widely used kernels is the **Radial Basis Function kernel** (**RBF kernel**) or Gaussian kernel:

$$k\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\frac{\left\|x^{(i)} - x^{(j)}\right\|^2}{2\sigma^2}\right)$$

This is often simplified to:

$$k\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\gamma \left\|x^{(i)} - x^{(j)}\right\|^2\right)$$
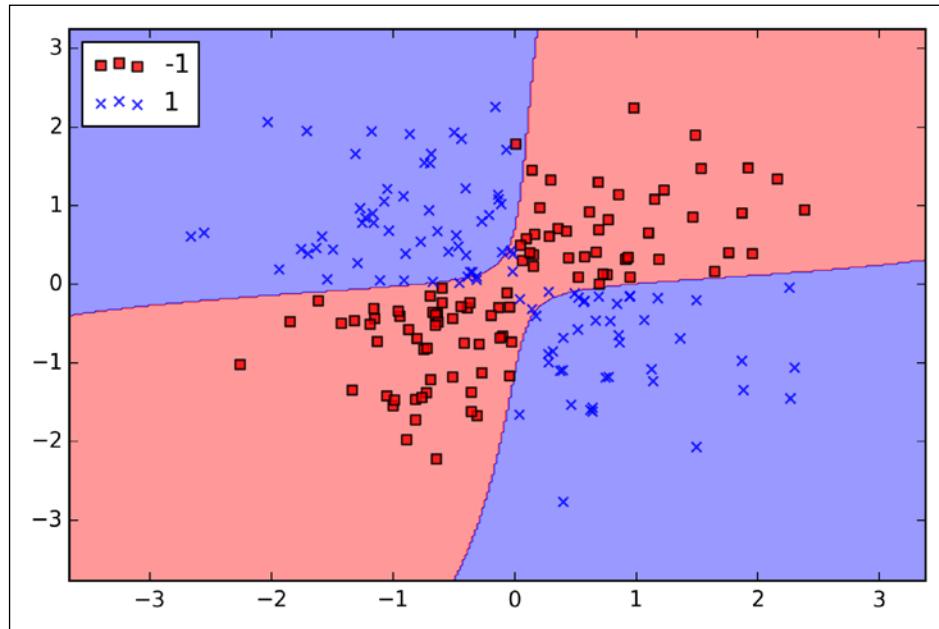
Here, $\gamma = \frac{1}{2\sigma^2}$ is a free parameter that is to be optimized.

Roughly speaking, the term *kernel* can be interpreted as a *similarity function* between a pair of samples. The minus sign inverts the distance measure into a similarity score and, due to the exponential term, the resulting similarity score will fall into a range between 1 (for exactly similar samples) and 0 (for very dissimilar samples).

Now that we defined the big picture behind the kernel trick, let's see if we can train a kernel SVM that is able to draw a nonlinear decision boundary that separates the XOR data well. Here, we simply use the SVC class from scikit-learn that we imported earlier and replace the parameter `kernel='linear'` with `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.show()
```

As we can see in the resulting plot, the kernel SVM separates the XOR data relatively well:

The γ parameter, which we set to `gamma=0.1`, can be understood as a *cut-off* parameter for the Gaussian sphere. If we increase the value for γ, we increase the influence or reach of the training samples, which leads to a softer decision boundary. To get a better intuition for γ, let's apply RBF kernel SVM to our Iris flower dataset:
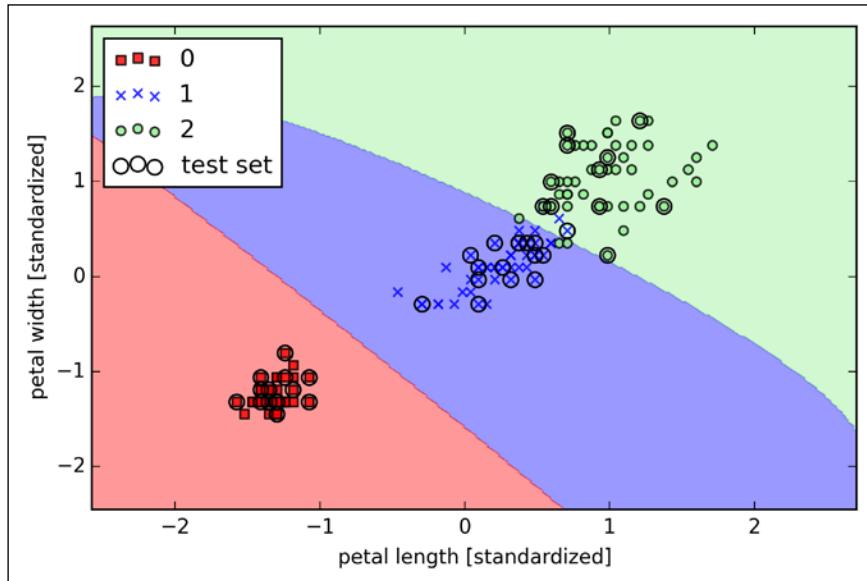
```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
```

```
...                        y_combined, classifier=svm,
...                        test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Since we chose a relatively small value for γ, the resulting decision boundary of the RBF kernel SVM model will be relatively soft, as shown in the following figure:
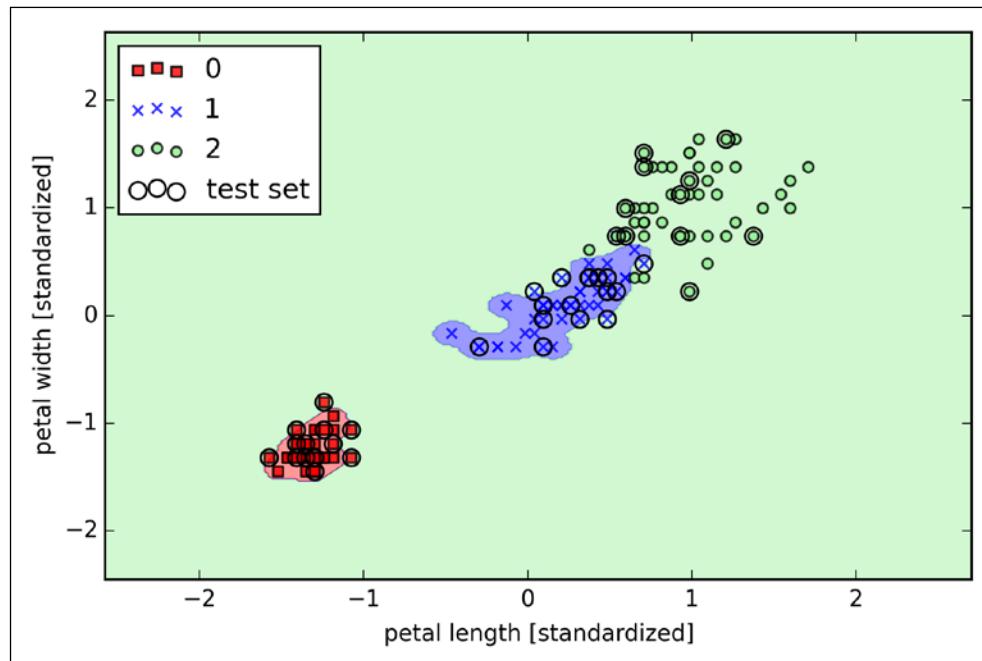
Now let's increase the value of γ and observe the effect on the decision boundary:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                        y_combined, classifier=svm,
...                        test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

In the resulting plot, we can now see that the decision boundary around the classes 0 and 1 is much tighter using a relatively large value of γ :
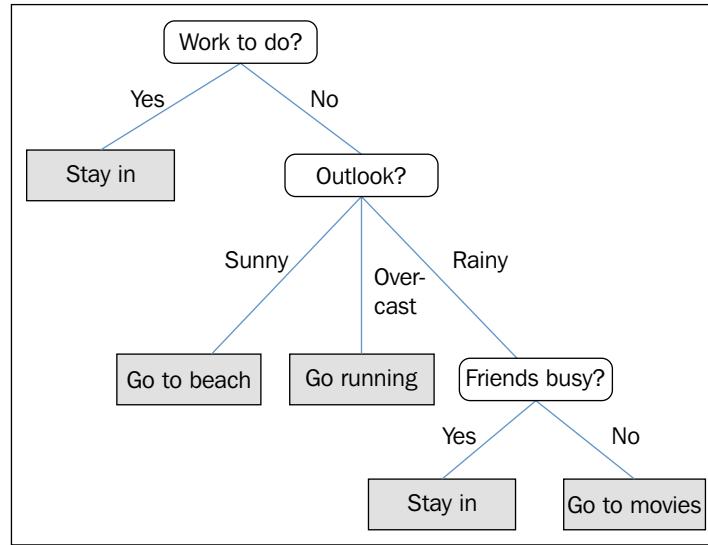
Although the model fits the training dataset very well, such a classifier will likely have a high generalization error on unseen data, which illustrates that the optimization of γ also plays an important role in controlling overfitting.

Decision tree learning

Decision tree classifiers are attractive models if we care about interpretability. Like the name *decision tree* suggests, we can think of this model as breaking down our data by making decisions based on asking a series of questions.

Let's consider the following example where we use a decision tree to decide upon an activity on a particular day:

Based on the features in our training set, the decision tree model learns a series of questions to infer the class labels of the samples. Although the preceding figure illustrated the concept of a decision tree based on categorical variables, the same concept applies if our features are real numbers like in the Iris dataset. For example, we could simply define a cut-off value along the **sepal width** feature axis and ask a binary question "sepal width ≥ 2.8 cm?"

Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **information gain** (**IG**), which will be explained in more detail in the following section. In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure. This means that the samples at each node all belong to the same class. In practice, this can result in a very deep tree with many nodes, which can easily lead to overfitting. Thus, we typically want to *prune* the tree by setting a limit for the maximal depth of the tree.

Maximizing information gain – getting the most bang for the buck

In order to split the nodes at the most informative features, we need to define an objective function that we want to optimize via the tree learning algorithm. Here, our objective function is to maximize the information gain at each split, which we define as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

Here, *f* is the feature to perform the split, D_p and D_j are the dataset of the parent and *j*th child node, *I* is our impurity measure, N_p is the total number of samples at the parent node, and N_j is the number of samples in the *j*th child node. As we can see, the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities—the lower the impurity of the child nodes, the larger the information gain. However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees. This means that each parent node is split into two child nodes, D_{left} and D_{right}:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

Now, the three impurity measures or splitting criteria that are commonly used in binary decision trees are **Gini impurity** (I_G), **entropy** (I_H), and the **classification error** (I_E). Let's start with the definition of entropy for all **non-empty** classes $p(i|t) \neq 0$:

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$$

Here, $p(i|t)$ is the proportion of the samples that belongs to class *i* for a particular node *t*. The entropy is therefore 0 if all samples at a node belong to the same class, and the entropy is maximal if we have a uniform class distribution. For example, in a binary class setting, the entropy is 0 if $p(i=1|t)=1$ or $p(i=0|t)=0$. If the classes are distributed uniformly with $p(i=1|t)=0.5$ and $p(i=0|t)=0.5$, the entropy is 1. Therefore, we can say that the entropy criterion attempts to maximize the mutual information in the tree.

Intuitively, the Gini impurity can be understood as a criterion to minimize the probability of misclassification:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(1-p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

Similar to entropy, the Gini impurity is maximal if the classes are perfectly mixed, for example, in a binary class setting ($c=2$):
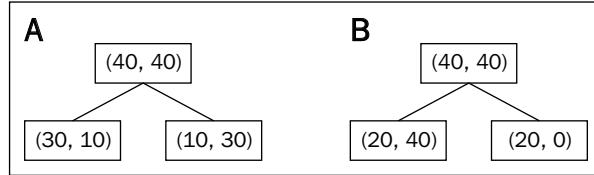
$$1 - \sum_{i=1}^{c} 0.5^2 = 0.5$$

However, in practice both the Gini impurity and entropy typically yield very similar results and it is often not worth spending much time on evaluating trees using different impurity criteria rather than experimenting with different pruning cut-offs.

Another impurity measure is the classification error:

$$I_E = 1 - \max\{p(i|t)\}$$

This is a useful criterion for pruning but not recommended for growing a decision tree, since it is less sensitive to changes in the class probabilities of the nodes. We can illustrate this by looking at the two possible splitting scenarios shown in the following figure:

We start with a dataset D_p at the parent node D_p that consists of 40 samples from class 1 and 40 samples from class 2 that we split into two datasets D_{left} and D_{right}, respectively. The information gain using the classification error as a splitting criterion would be the same ($IG_E = 0.25$) in both scenario A and B:

$$I_E\left(D_p\right) = 1 - 0.5 = 0.5$$

$$A : I_E\left(D_{left}\right) = 1 - \frac{3}{4} = 0.25$$

$$A : I_E\left(D_{right}\right) = 1 - \frac{3}{4} = 0.25$$

$$A : IG_E = 0.5 - \frac{4}{8}0.25 - \frac{4}{8}0.25 = 0.25$$

$$B : I_E\left(D_{left}\right) = 1 - \frac{4}{6} = \frac{1}{3}$$

$$B : I_E\left(D_{right}\right) = 1 - 1 = 0$$

$$B : IG_E = 0.5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0.25$$

However, the Gini impurity would favor the split in scenario $B\left(IG_G = 0.1\overline{6}\right)$ over scenario $A\left(IG_G = 0.125\right)$, which is indeed more *pure*:

$$I_G\left(D_p\right) = 1 - \left(0.5^2 + 0.5^2\right) = 0.5$$

$$A : I_G\left(D_{left}\right) = 1 - \left(\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A : I_G\left(D_{right}\right) = 1 - \left(\left(\frac{1}{4}\right)^2 + \left(\frac{3}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A : IG_G = 0.5 - \frac{4}{8}0.375 - \frac{4}{8}0.375 = 0.125$$

$$B : I_G\left(D_{left}\right) = 1 - \left(\left(\frac{2}{6}\right)^2 + \left(\frac{4}{6}\right)^2\right) = \frac{4}{9} = 0.\overline{4}$$

$$B : I_G\left(D_{right}\right) = 1 - \left(1^2 + 0^2\right) = 0$$

$$B : IG_G = 0.5 - \frac{6}{8}0.\overline{4} - 0 = 0.1\overline{6}$$

Similarly, the entropy criterion would favor scenario $B\left(IG_H = 0.31\right)$ over scenario $A\left(IG_H = 0.19\right)$:

$$I_H\left(D_p\right) = -\left(0.5 \, \log_2\left(0.5\right) + 0.5 \, \log_2\left(0.5\right)\right) = 1$$

$$A : I_H\left(D_{left}\right) = -\left(\frac{3}{4}\log_2\left(\frac{3}{4}\right) + \frac{1}{4}\log_2\left(\frac{1}{4}\right)\right) = 0.81$$

$$A : I_H \left(D_{right} \right) = - \left(\frac{1}{4} \log_2 \left(\frac{1}{4} \right) + \frac{3}{4} \log_2 \left(\frac{3}{4} \right) \right) = 0.81$$

$$A : IG_H = 1 - \frac{4}{8} 0.81 - \frac{4}{8} 0.81 = 0.19$$

$$B : I_H \left(D_{left} \right) = - \left(\frac{2}{6} \log_2 \left(\frac{2}{6} \right) + \frac{4}{6} \log_2 \left(\frac{4}{6} \right) \right) = 0.92$$

$$B : I_H \left(D_{right} \right) = 0$$
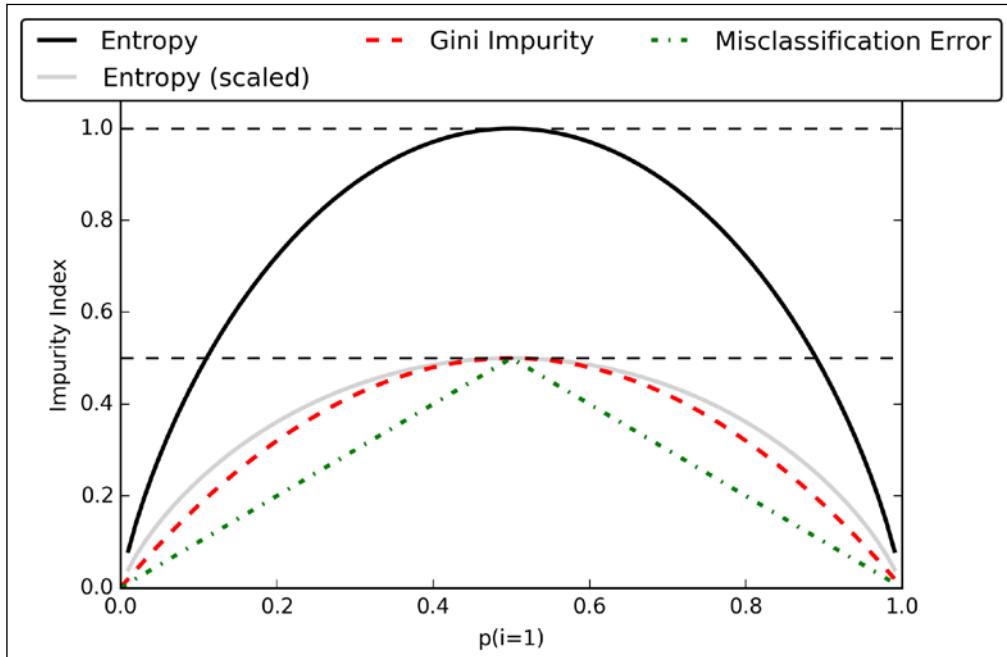
$$B : IG_H = 1 - \frac{6}{8} 0.92 - 0 = 0.31$$

For a more visual comparison of the three different impurity criteria that we discussed previously, let's plot the impurity indices for the probability range [0, 1] for class 1. Note that we will also add in a scaled version of the entropy (*entropy/2*) to observe that the Gini impurity is an intermediate measure between entropy and the classification error. The code is as follows:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                 ['Entropy', 'Entropy (scaled)',
...                 'Gini Impurity',
```

```
...                             'Misclassification Error'],
...                             ['-', '-', '--', '-.'],
...                             ['black', 'lightgray',
...                                 'red', 'green', 'cyan']):
...         line = ax.plot(x, i, label=lab,
...                             linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...             ncol=3, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('Impurity Index')
>>> plt.show()
```

The plot produced by the preceding code example is as follows:
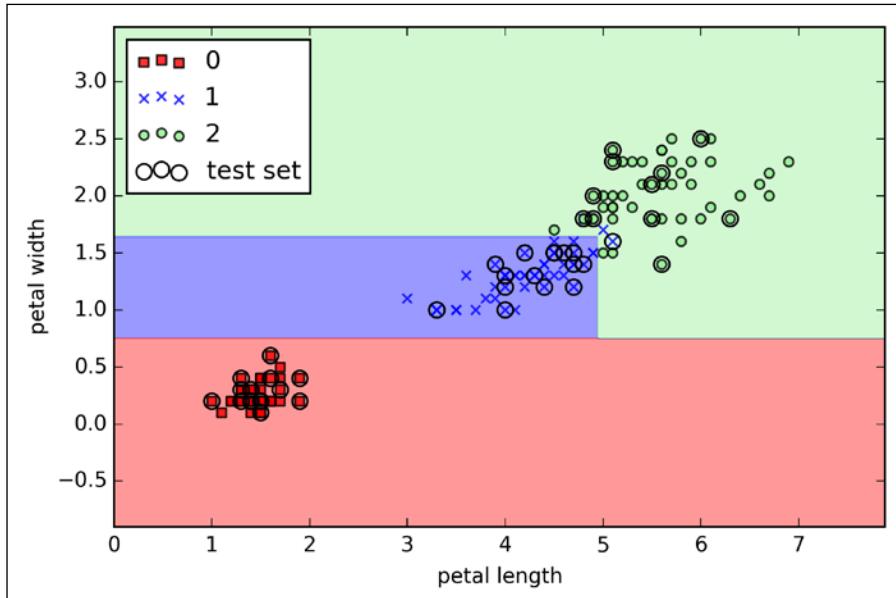
Building a decision tree

Decision trees can build complex decision boundaries by dividing the feature space into rectangles. However, we have to be careful since the deeper the decision tree, the more complex the decision boundary becomes, which can easily result in overfitting. Using scikit-learn, we will now train a decision tree with a maximum depth of 3 using entropy as a criterion for impurity. Although feature scaling may be desired for visualization purposes, note that feature scaling is not a requirement for decision tree algorithms. The code is as follows:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=3, random_state=0)
>>> tree.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined, y_combined,
...                       classifier=tree, test_idx=range(105,150))
>>>plt.xlabel('petal length [cm]')
>>>plt.ylabel('petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we get the typical axis-parallel decision boundaries of the decision tree:
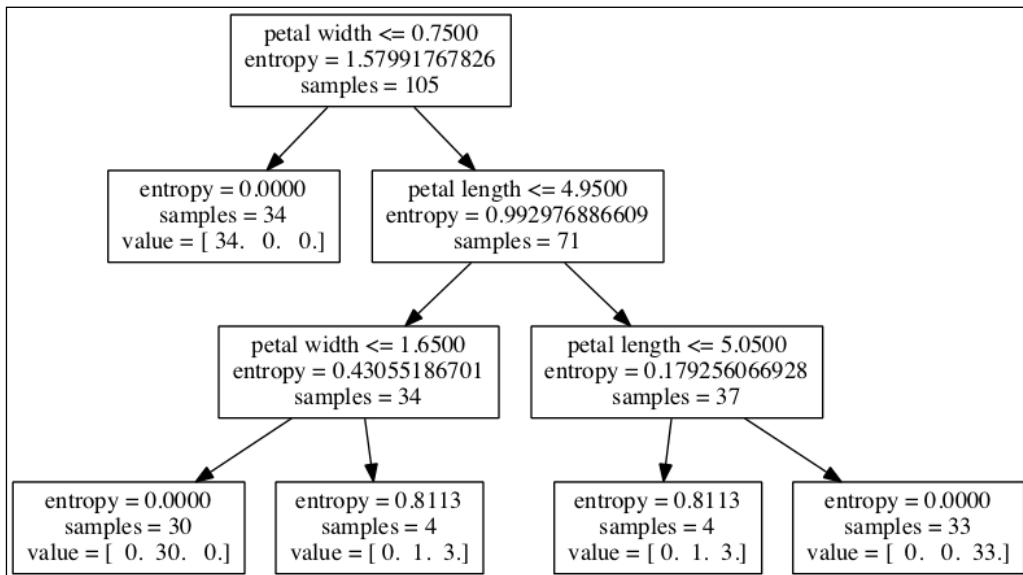
A nice feature in scikit-learn is that it allows us to export the decision tree as a `.dot` file after training, which we can visualize using the GraphViz program. This program is freely available at `http://www.graphviz.org` and supported by Linux, Windows, and Mac OS X.

First, we create the `.dot` file via scikit-learn using the `export_graphviz` function from the `tree` submodule, as follows:

```
>>> from sklearn.tree import export_graphviz
>>> export_graphviz(tree,
...                  out_file='tree.dot',
...                  feature_names=['petal length', 'petal width'])
```

After we have installed GraphViz on our computer, we can convert the `tree.dot` file into a PNG file by executing the following command from the command line in the location where we saved the `tree.dot` file:

```
> dot -Tpng tree.dot -o tree.png
```


Looking at the decision tree figure that we created via GraphViz, we can now nicely trace back the splits that the decision tree determined from our training dataset. We started with 105 samples at the root and split it into two child nodes with 34 and 71 samples each using the **petal width** cut-off ≤ 0.75 cm. After the first split, we can see that the left child node is already pure and only contains samples from the Iris-Setosa class (entropy = 0). The further splits on the right are then used to separate the samples from the Iris-Versicolor and Iris-Virginica classes.

Combining weak to strong learners via random forests

Random forests have gained huge popularity in applications of machine learning during the last decade due to their good classification performance, scalability, and ease of use. Intuitively, a random forest can be considered as an *ensemble* of decision trees. The idea behind ensemble learning is to combine **weak learners** to build a more robust model, a **strong learner**, that has a better generalization error and is less susceptible to overfitting. The random forest algorithm can be summarized in four simple steps:

1. Draw a random **bootstrap** sample of size n (randomly choose n samples from the training set with replacement).

2. Grow a decision tree from the bootstrap sample. At each node:

    1. Randomly select d features without replacement.

    2. Split the node using the feature that provides the best split according to the objective function, for instance, by maximizing the information gain.

3. Repeat the steps 1 to 2 k times.

4. Aggregate the prediction by each tree to assign the class label by **majority vote**. Majority voting will be discussed in more detail in *Chapter 7*, *Combining Different Models for Ensemble Learning*.

There is a slight modification in step 2 when we are training the individual decision trees: instead of evaluating all features to determine the best split at each node, we only consider a random subset of those.
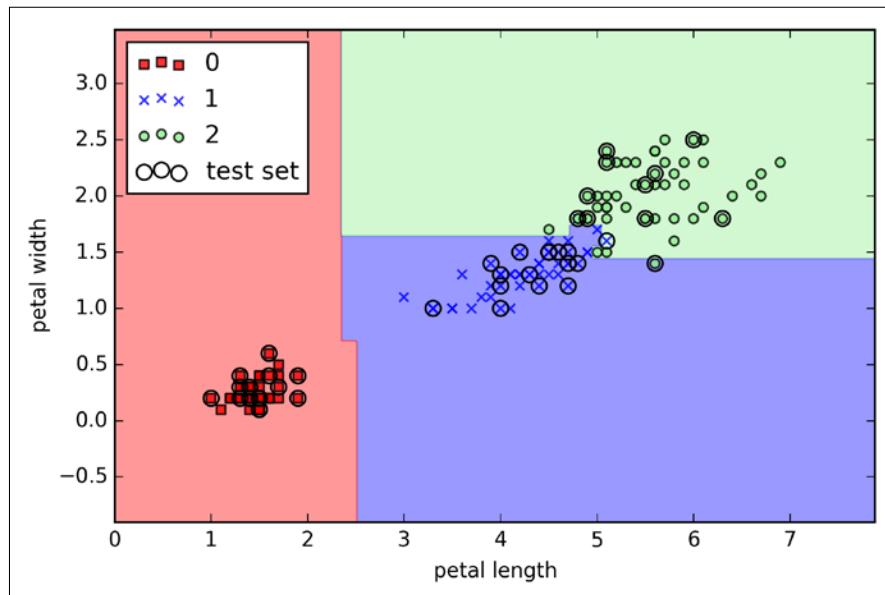
Although random forests don't offer the same level of interpretability as decision trees, a big advantage of random forests is that we don't have to worry so much about choosing good hyperparameter values. We typically don't need to prune the random forest since the ensemble model is quite robust to noise from the individual decision trees. The only parameter that we really need to care about in practice is the number of trees k (step 3) that we choose for the random forest. Typically, the larger the number of trees, the better the performance of the random forest classifier at the expense of an increased computational cost.

Although it is less common in practice, other hyperparameters of the random forest classifier that can be optimized—using techniques we will discuss in *Chapter 5*, *Compressing Data via Dimensionality Reduction*—are the size *n* of the bootstrap sample (step 1) and the number of features *d* that is randomly chosen for each split (step 2.1), respectively. Via the sample size *n* of the bootstrap sample, we control the bias-variance tradeoff of the random forest. By choosing a larger value for *n*, we decrease the randomness and thus the forest is more likely to overfit. On the other hand, we can reduce the degree of overfitting by choosing smaller values for *n* at the expense of the model performance. In most implementations, including the `RandomForestClassifier` implementation in scikit-learn, the sample size of the bootstrap sample is chosen to be equal to the number of samples in the original training set, which usually provides a good bias-variance tradeoff. For the number of features *d* at each split, we want to choose a value that is smaller than the total number of features in the training set. A reasonable default that is used in scikit-learn and other implementations is $d = \sqrt{m}$, where *m* is the number of features in the training set.

Conveniently, we don't have to construct the random forest classifier from individual decision trees by ourselves; there is already an implementation in scikit-learn that we can use:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='entropy',
...                                 n_estimators=10,
...                                 random_state=1,
...                                 n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                 classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('petal length')
>>> plt.ylabel('petal width')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code, we should see the decision regions formed by the ensemble of trees in the random forest, as shown in the following figure:

Using the preceding code, we trained a random forest from 10 decision trees via the `n_estimators` parameter and used the entropy criterion as an impurity measure to split the nodes. Although we are growing a very small random forest from a very small training dataset, we used the `n_jobs` parameter for demonstration purposes, which allows us to parallelize the model training using multiple cores of our computer (here, two).

K-nearest neighbors – a lazy learning algorithm

The last supervised learning algorithm that we want to discuss in this chapter is the **k-nearest neighbor classifier** (**KNN**), which is particularly interesting because it is fundamentally different from the learning algorithms that we have discussed so far.

KNN is a typical example of a **lazy learner**. It is called *lazy* not because of its apparent simplicity, but because it doesn't learn a discriminative function from the training data but memorizes the training dataset instead.

Parametric versus nonparametric models

Machine learning algorithms can be grouped into **parametric** and **nonparametric** models. Using parametric models, we estimate parameters from the training dataset to learn a function that can classify new data points without requiring the original training dataset anymore. Typical examples of parametric models are the perceptron, logistic regression, and the linear SVM. In contrast, nonparametric models can't be characterized by a fixed set of parameters, and the number of parameters grows with the training data. Two examples of nonparametric models that we have seen so far are the decision tree classifier/random forest and the kernel SVM.
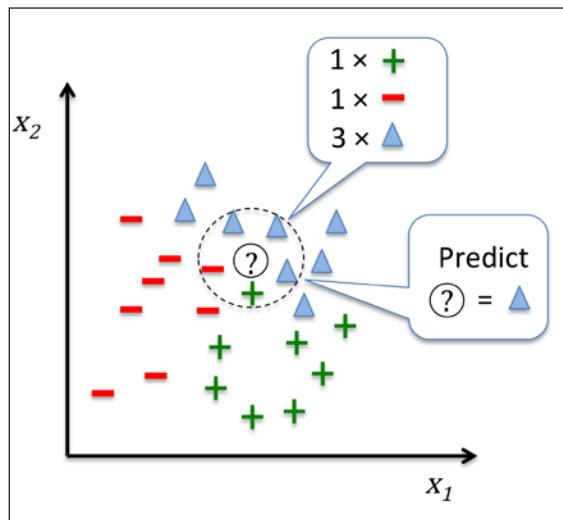
KNN belongs to a subcategory of nonparametric models that is described as **instance-based learning**. Models based on instance-based learning are characterized by memorizing the training dataset, and lazy learning is a special case of instance-based learning that is associated with no (zero) cost during the learning process.

The KNN algorithm itself is fairly straightforward and can be summarized by the following steps:

1. Choose the number of *k* and a distance metric.
2. Find the *k* nearest neighbors of the sample that we want to classify.
3. Assign the class label by majority vote.

The following figure illustrates how a new data point (**?**) is assigned the triangle class label based on majority voting among its five nearest neighbors.
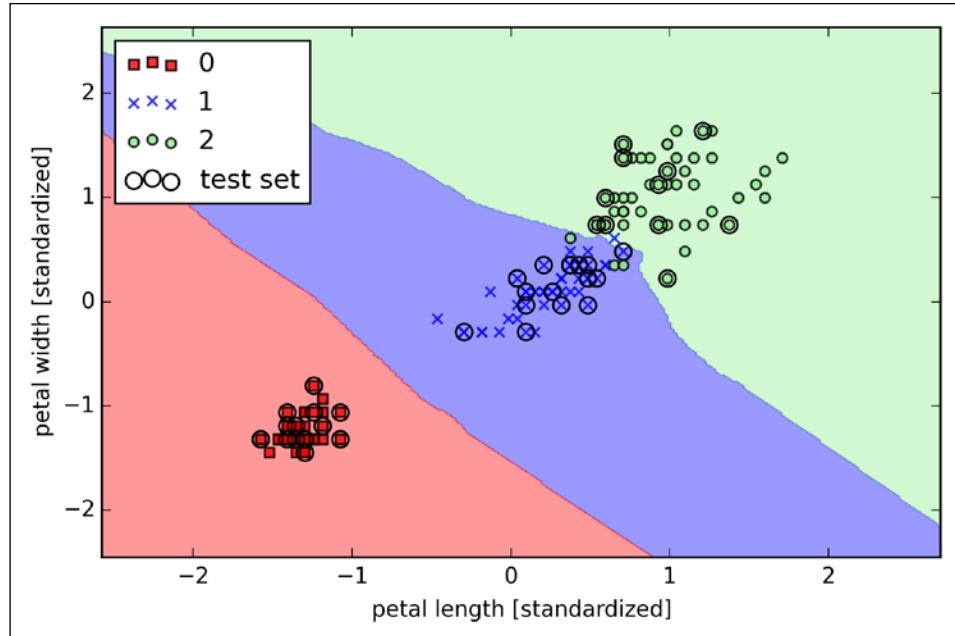
Based on the chosen distance metric, the KNN algorithm finds the *k* samples in the training dataset that are closest (most similar) to the point that we want to classify. The class label of the new data point is then determined by a majority vote among its *k* nearest neighbors.

The main advantage of such a memory-based approach is that the classifier immediately adapts as we collect new training data. However, the downside is that the computational complexity for classifying new samples grows linearly with the number of samples in the training dataset in the worst-case scenario—unless the dataset has very few dimensions (features) and the algorithm has been implemented using efficient data structures such as KD-trees. (J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS), 3(3):209–226, 1977.) Furthermore, we can't discard training samples since no *training* step is involved. Thus, storage space can become a challenge if we are working with large datasets.

By executing the following code, we will now implement a KNN model in scikit-learn using an Euclidean distance metric:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                            metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.show()
```

By specifying five neighbors in the KNN model for this dataset, we obtain a relatively smooth decision boundary, as shown in the following figure:

> In the case of a tie, the scikit-learn implementation of the KNN algorithm will prefer the neighbors with a closer distance to the sample. If the neighbors have a similar distance, the algorithm will choose the class label that comes first in the training dataset.

The *right* choice of *k* is crucial to find a good balance between over- and underfitting. We also have to make sure that we choose a distance metric that is appropriate for the features in the dataset. Often, a simple Euclidean distance measure is used for real-valued samples, for example, the flowers in our Iris dataset, which have features measured in centimeters. However, if we are using a Euclidean distance measure, it is also important to standardize the data so that each feature contributes equally to the distance. The `'minkowski'` distance that we used in the previous code is just a generalization of the Euclidean and Manhattan distance that can be written as follows:

$$d\left(x^{(i)}, x^{(j)}\right) = \sqrt[p]{\sum_k \left| x_k^{(i)} - x_k^{(j)} \right|^p}$$

It becomes the Euclidean distance if we set the parameter `p=2` or the Manhattan distance at `p=1`, respectively. Many other distance metrics are available in scikit-learn and can be provided to the `metric` parameter. They are listed at `http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html`.

> **The curse of dimensionality**
>
> It is important to mention that KNN is very susceptible to overfitting due to the **curse of dimensionality**. The curse of dimensionality describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset. Intuitively, we can think of even the closest neighbors being too far away in a high-dimensional space to give a good estimate.
>
> We have discussed the concept of regularization in the section about logistic regression as one way to avoid overfitting. However, in models where regularization is not applicable such as decision trees and KNN, we can use feature selection and dimensionality reduction techniques to help us avoid the curse of dimensionality. This will be discussed in more detail in the next chapter.

Summary

In this chapter, you learned about many different machine algorithms that are used to tackle linear and nonlinear problems. We have seen that decision trees are particularly attractive if we care about interpretability. Logistic regression is not only a useful model for online learning via stochastic gradient descent, but also allows us to predict the probability of a particular event. Although support vector machines are powerful linear models that can be extended to nonlinear problems via the kernel trick, they have many parameters that have to be tuned in order to make good predictions. In contrast, ensemble methods such as random forests don't require much parameter tuning and don't overfit so easily as decision trees, which makes it an attractive model for many practical problem domains. The K-nearest neighbor classifier offers an alternative approach to classification via lazy learning that allows us to make predictions without any model training but with a more computationally expensive prediction step.

However, even more important than the choice of an appropriate learning algorithm is the available data in our training dataset. No algorithm will be able to make good predictions without informative and discriminatory features.

In the next chapter, we will discuss important topics regarding the preprocessing of data, feature selection, and dimensionality reduction, which we will need to build powerful machine learning models. Later in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will see how we can evaluate and compare the performance of our models and learn useful tricks to fine-tune the different algorithms.

4
Building Good Training Sets – Data Preprocessing

The quality of the data and the amount of useful information that it contains are key factors that determine how well a machine learning algorithm can learn. Therefore, it is absolutely critical that we make sure to examine and preprocess a dataset before we feed it to a learning algorithm. In this chapter, we will discuss the essential data preprocessing techniques that will help us to build good machine learning models.

The topics that we will cover in this chapter are as follows:

- Removing and imputing missing values from the dataset
- Getting categorical data into shape for machine learning algorithms
- Selecting relevant features for the model construction

Dealing with missing data

It is not uncommon in real-world applications that our samples are missing one or more values for various reasons. There could have been an error in the data collection process, certain measurements are not applicable, particular fields could have been simply left blank in a survey, for example. We typically see *missing values* as the blank spaces in our data table or as placeholder strings such as `NaN` (Not A Number).

Unfortunately, most computational tools are unable to handle such missing values or would produce unpredictable results if we simply ignored them. Therefore, it is crucial that we take care of those missing values before we proceed with further analyses. But before we discuss several techniques for dealing with missing values, let's create a simple example data frame from a **CSV** (**comma-separated values**) file to get a better grasp of the problem:

```
>>> import pandas as pd
>>> from io import StringIO
>>> csv_data = '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''
>>> # If you are using Python 2.7, you need
>>> # to convert the string to unicode:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
    A   B   C   D
0   1   2   3   4
1   5   6 NaN   8
2  10  11  12 NaN
```

Using the preceding code, we read CSV-formatted data into a pandas `DataFrame` via the `read_csv` function and noticed that the two missing cells were replaced by `NaN`. The `StringIO` function in the preceding code example was simply used for the purposes of illustration. It allows us to read the string assigned to `csv_data` into a pandas `DataFrame` as if it was a regular CSV file on our hard drive.

For a larger `DataFrame`, it can be tedious to look for missing values manually; in this case, we can use the `isnull` method to return a `DataFrame` with Boolean values that indicate whether a cell contains a numeric value (`False`) or if data is missing (`True`). Using the `sum` method, we can then return the number of missing values per column as follows:

```
>>> df.isnull().sum()
A    0
B    0
C    1
D    1
dtype: int64
```

This way, we can count the number of missing values per column; in the following subsections, we will take a look at different strategies for how to deal with this missing data.

> Although scikit-learn was developed for working with NumPy arrays, it can sometimes be more convenient to preprocess data using pandas' `DataFrame`. We can always access the underlying NumPy array of the `DataFrame` via the `values` attribute before we feed it into a scikit-learn estimator:
>
> ```
> >>> df.values
> array([[  1.,    2.,    3.,    4.],
>        [  5.,    6.,   nan,    8.],
>        [ 10.,   11.,   12.,   nan]])
> ```

Eliminating samples or features with missing values

One of the easiest ways to deal with missing data is to simply remove the corresponding features (columns) or samples (rows) from the dataset entirely; rows with missing values can be easily dropped via the `dropna` method:

```
>>> df.dropna()
   A  B  C  D
0  1  2  3  4
```

Similarly, we can drop columns that have at least one `NaN` in any row by setting the `axis` argument to `1`:

```
>>> df.dropna(axis=1)
    A   B
0   1   2
1   5   6
2  10  11
```

The `dropna` method supports several additional parameters that can come in handy:

```
# only drop rows where all columns are NaN
>>> df.dropna(how='all')

# drop rows that have not at least 4 non-NaN values
>>> df.dropna(thresh=4)

# only drop rows where NaN appear in specific columns (here: 'C')
>>> df.dropna(subset=['C'])
```

Although the removal of missing data seems to be a convenient approach, it also comes with certain disadvantages; for example, we may end up removing too many samples, which will make a reliable analysis impossible. Or, if we remove too many feature columns, we will run the risk of losing valuable information that our classifier needs to discriminate between classes. In the next section, we will thus look at one of the most commonly used alternatives for dealing with missing values: interpolation techniques.

Imputing missing values
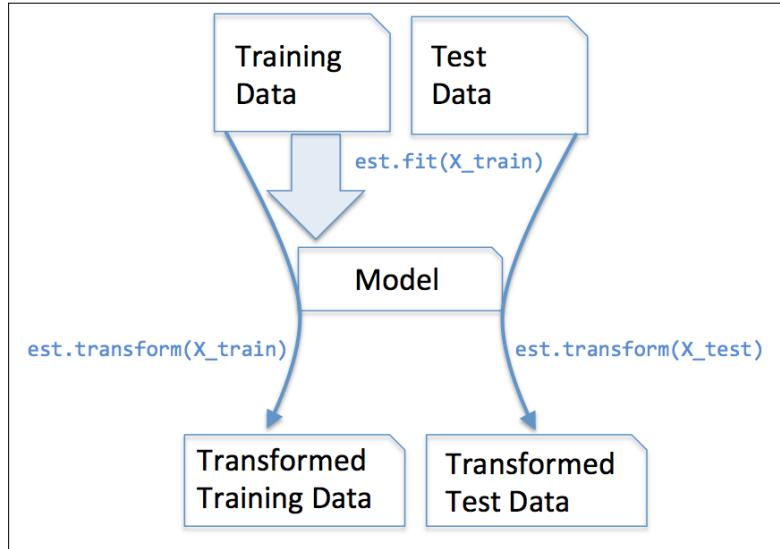
Often, the removal of samples or dropping of entire feature columns is simply not feasible, because we might lose too much valuable data. In this case, we can use different interpolation techniques to estimate the missing values from the other training samples in our dataset. One of the most common interpolation techniques is **mean imputation**, where we simply replace the missing value by the mean value of the entire feature column. A convenient way to achieve this is by using the `Imputer` class from scikit-learn, as shown in the following code:

```
>>> from sklearn.preprocessing import Imputer
>>> imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imr = imr.fit(df)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[  1.,    2.,    3.,    4.],
       [  5.,    6.,   7.5,    8.],
       [ 10.,   11.,   12.,    6.]])
```
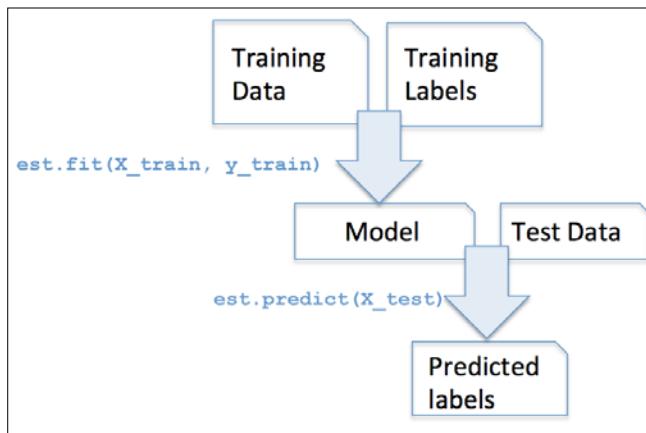
Here, we replaced each `NaN` value by the corresponding mean, which is separately calculated for each feature column. If we changed the setting `axis=0` to `axis=1`, we'd calculate the row means. Other options for the `strategy` parameter are `median` or `most_frequent`, where the latter replaces the missing values by the most frequent values. This is useful for imputing categorical feature values.

Understanding the scikit-learn estimator API

In the previous section, we used the `Imputer` class from scikit-learn to impute missing values in our dataset. The `Imputer` class belongs to the so-called **transformer** classes in scikit-learn that are used for data transformation. The two essential methods of those estimators are `fit` and `transform`. The `fit` method is used to learn the parameters from the training data, and the `transform` method uses those parameters to transform the data. Any data array that is to be transformed needs to have the same number of features as the data array that was used to fit the model. The following figure illustrates how a transformer fitted on the training data is used to transform a training dataset as well as a new test dataset:

The classifiers that we used in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, belong to the so-called estimators in scikit-learn with an API that is conceptually very similar to the transformer class. Estimators have a `predict` method but can also have a `transform` method, as we will see later. As you may recall, we also used the `fit` method to learn the parameters of a model when we trained those estimators for classification. However, in supervised learning tasks, we additionally provide the class labels for fitting the model, which can then be used to make predictions about new data samples via the `predict` method, as illustrated in the following figure:

Handling categorical data

So far, we have only been working with numerical values. However, it is not uncommon that real-world datasets contain one or more categorical feature columns. When we are talking about categorical data, we have to further distinguish between **nominal** and **ordinal** features. Ordinal features can be understood as categorical values that can be sorted or ordered. For example, *T-shirt size* would be an ordinal feature, because we can define an order $XL > L > M$. In contrast, nominal features don't imply any order and, to continue with the previous example, we could think of *T-shirt color* as a nominal feature since it typically doesn't make sense to say that, for example, *red* is larger than *blue*.

Before we explore different techniques to handle such categorical data, let's create a new data frame to illustrate the problem:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...            ['green', 'M', 10.1, 'class1'],
...            ['red', 'L', 13.5, 'class2'],
...            ['blue', 'XL', 15.3, 'class1']])
>>> df.columns = ['color', 'size', 'price', 'classlabel']
>>> df
   color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1
```

As we can see in the preceding output, the newly created `DataFrame` contains a nominal feature (`color`), an ordinal feature (`size`), and a numerical feature (`price`) column. The class labels (assuming that we created a dataset for a supervised learning task) are stored in the last column. The learning algorithms for classification that we discuss in this book do not use ordinal information in class labels.

Mapping ordinal features

To make sure that the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers. Unfortunately, there is no convenient function that can automatically derive the correct order of the labels of our `size` feature. Thus, we have to define the mapping manually. In the following simple example, let's assume that we know the difference between features, for example, $XL = L + 1 = M + 2$.

```
>>> size_mapping = {
...                 'XL': 3,
...                 'L': 2,
```

```
...                    'M': 1}
>>> df['size'] = df['size'].map(size_mapping)
>>> df
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

If we want to transform the integer values back to the original string representation at a later stage, we can simply define a reverse-mapping dictionary `inv_size_mapping = {v: k for k, v in size_mapping.items()}` that can then be used via the pandas' `map` method on the transformed feature column similar to the `size_mapping` dictionary that we used previously.

Encoding class labels

Many machine learning libraries require that class labels are encoded as integer values. Although most estimators for classification in scikit-learn convert class labels to integers internally, it is considered good practice to provide class labels as integer arrays to avoid technical glitches. To encode the class labels, we can use an approach similar to the mapping of ordinal features discussed previously. We need to remember that class labels are *not* ordinal, and it doesn't matter which integer number we assign to a particular string-label. Thus, we can simply enumerate the class labels starting at 0:

```
>>> import numpy as np
>>> class_mapping = {label:idx for idx,label in
...                    enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

Next we can use the mapping dictionary to transform the class labels into integers:

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
   color  size  price  classlabel
0  green     1   10.1           0
1    red     2   13.5           1
2   blue     3   15.3           0
```

We can reverse the key-value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation:

```
>>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>>> df
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

Alternatively, there is a convenient `LabelEncoder` class directly implemented in scikit-learn to achieve the same:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([0, 1, 0])
```

Note that the `fit_transform` method is just a shortcut for calling `fit` and `transform` separately, and we can use the `inverse_transform` method to transform the integer class labels back into their original string representation:

```
>>> class_le.inverse_transform(y)
array(['class1', 'class2', 'class1'], dtype=object)
```

Performing one-hot encoding on nominal features

In the previous section, we used a simple dictionary-mapping approach to convert the ordinal size feature into integers. Since scikit-learn's estimators treat class labels without any order, we used the convenient `LabelEncoder` class to encode the string labels into integers. It may appear that we could use a similar approach to transform the nominal `color` column of our dataset, as follows:

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

After executing the preceding code, the first column of the NumPy array x now holds the new `color` values, which are encoded as follows:

- blue → 0
- green → 1
- red → 2

If we stop at this point and feed the array to our classifier, we will make one of the most common mistakes in dealing with categorical data. Can you spot the problem? Although the color values don't come in any particular order, a learning algorithm will now assume that *green* is larger than *blue*, and *red* is larger than *green*. Although this assumption is incorrect, the algorithm could still produce useful results. However, those results would not be optimal.

A common workaround for this problem is to use a technique called **one-hot encoding**. The idea behind this approach is to create a new **dummy feature** for each unique value in the nominal feature column. Here, we would convert the `color` feature into three new features: `blue`, `green`, and `red`. Binary values can then be used to indicate the particular color of a sample; for example, a blue sample can be encoded as `blue=1, green=0, red=0`. To perform this transformation, we can use the `OneHotEncoder` that is implemented in the `scikit-learn.preprocessing` module:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> ohe = OneHotEncoder(categorical_features=[0])
>>> ohe.fit_transform(X).toarray()
array([[ 0. ,   1. ,   0. ,   1. ,  10.1],
       [ 0. ,   0. ,   1. ,   2. ,  13.5],
       [ 1. ,   0. ,   0. ,   3. ,  15.3]])
```

When we initialized the `OneHotEncoder`, we defined the column position of the variable that we want to transform via the `categorical_features` parameter (note that `color` is the first column in the feature matrix X). By default, the `OneHotEncoder` returns a sparse matrix when we use the `transform` method, and we converted the sparse matrix representation into a regular (*dense*) NumPy array for the purposes of visualization via the `toarray` method. Sparse matrices are simply a more efficient way of storing large datasets, and one that is supported by many scikit-learn functions, which is especially useful if it contains a lot of zeros. To omit the `toarray` step, we could initialize the encoder as `OneHotEncoder(…, sparse=False)` to return a regular NumPy array.

An even more convenient way to create those dummy features via one-hot encoding is to use the `get_dummies` method implemented in pandas. Applied on a `DataFrame`, the `get_dummies` method will only convert string columns and leave all other columns unchanged:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
   price  size  color_blue  color_green  color_red
0   10.1    1           0            1          0
1   13.5    2           0            0          1
2   15.3    3           1            0          0
```

Partitioning a dataset in training and test sets

We briefly introduced the concept of partitioning a dataset into separate datasets for training and testing in *Chapter 1*, *Giving Computers the Ability to Learn from Data*, and *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Remember that the test set can be understood as the *ultimate test* of our model before we let it loose on the real world. In this section, we will prepare a new dataset, the **Wine** dataset. After we have preprocessed the dataset, we will explore different techniques for feature selection to reduce the dimensionality of a dataset.

The Wine dataset is another open-source dataset that is available from the UCI machine learning repository (`https://archive.ics.uci.edu/ml/datasets/Wine`); it consists of 178 wine samples with 13 features describing their different chemical properties.

Using the pandas library, we will directly read in the open source Wine dataset from the UCI machine learning repository:

```
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash', 'Magnesium',
...                    'Total phenols', 'Flavanoids',
...                    'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>>  print('Class labels', np.unique(df_wine['Class label']))
Class labels [1 2 3]
>>> df_wine.head()
```

The 13 different features in the **Wine** dataset, describing the chemical properties of the 178 wine samples, are listed in the following table:

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

The samples belong to one of three different classes, 1, 2, and 3, which refer to the three different types of grapes that have been grown in different regions in Italy.

A convenient way to randomly partition this dataset into a separate *test* and *training* dataset is to use the `train_test_split` function from scikit-learn's `cross_validation` submodule:

```
>>> from sklearn.cross_validation import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...          train_test_split(X, y, test_size=0.3, random_state=0)
```

First, we assigned the NumPy array representation of feature columns 1-13 to the variable X, and we assigned the class labels from the first column to the variable y. Then, we used the `train_test_split` function to randomly split X and y into separate training and test datasets. By setting `test_size=0.3` we assigned 30 percent of the wine samples to X_test and y_test, and the remaining 70 percent of the samples were assigned to X_train and y_train, respectively.

> If we are dividing a dataset into training and test datasets, we have to keep in mind that we are withholding valuable information that the learning algorithm could benefit from. Thus, we don't want to allocate too much information to the test set. However, the smaller the test set, the more inaccurate the estimation of the generalization error. Dividing a dataset into training and test sets is all about balancing this trade-off. In practice, the most commonly used splits are 60:40, 70:30, or 80:20, depending on the size of the initial dataset. However, for large datasets, 90:10 or 99:1 splits into training and test subsets are also common and appropriate. Instead of discarding the allocated test data after model training and evaluation, it is a good idea to retrain a classifier on the entire dataset for optimal performance.

Bringing features onto the same scale

Feature scaling is a crucial step in our **preprocessing** pipeline that can easily be forgotten. Decision trees and random forests are one of the very few machine learning algorithms where we don't need to worry about feature scaling. However, the majority of machine learning and optimization algorithms behave much better if features are on the same scale, as we saw in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, when we implemented the **gradient descent** optimization algorithm.

The importance of feature scaling can be illustrated by a simple example. Let's assume that we have two features where one feature is measured on a scale from 1 to 10 and the second feature is measured on a scale from 1 to 100,000. When we think of the squared error function in **Adaline** in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, it is intuitive to say that the algorithm will mostly be busy optimizing the weights according to the larger errors in the second feature. Another example is the **k-nearest neighbors** (**KNN**) algorithm with a Euclidean distance measure; the computed distances between samples will be dominated by the second feature axis.

Now, there are two common approaches to bringing different features onto the same scale: **normalization** and **standardization**. Those terms are often used quite loosely in different fields, and the meaning has to be derived from the context. Most often, normalization refers to the rescaling of the features to a range of [0, 1], which is a special case of min-max scaling. To normalize our data, we can simply apply the min-max scaling to each feature column, where the new value $x_{norm}^{(i)}$ of a sample $x^{(i)}$ can be calculated as follows:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

Here, $x^{(i)}$ is a particular sample, x_{min} is the smallest value in a feature column, and x_{max} the largest value, respectively.

The min-max scaling procedure is implemented in scikit-learn and can be used as follows:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

Although normalization via min-max scaling is a commonly used technique that is useful when we need values in a bounded interval, standardization can be more practical for many machine learning algorithms. The reason is that many linear models, such as the logistic regression and SVM that we remember from *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, initialize the weights to 0 or small random values close to 0. Using standardization, we center the feature columns at mean 0 with standard deviation 1 so that the feature columns take the form of a normal distribution, which makes it easier to learn the weights. Furthermore, standardization maintains useful information about outliers and makes the algorithm less sensitive to them in contrast to min-max scaling, which scales the data to a limited range of values.

The procedure of standardization can be expressed by the following equation:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Here, μ_x is the sample mean of a particular feature column and σ_x the corresponding standard deviation, respectively.

The following table illustrates the difference between the two commonly used feature scaling techniques, standardization and normalization on a simple sample dataset consisting of numbers 0 to 5:

input	standardized	normalized
0.0	-1.336306	0.0
1.0	-0.801784	0.2
2.0	-0.267261	0.4
3.0	0.267261	0.6
4.0	0.801784	0.8
5.0	1.336306	1.0

Similar to `MinMaxScaler`, scikit-learn also implements a class for standardization:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

Again, it is also important to highlight that we fit the `StandardScaler` only once on the training data and use those parameters to transform the test set or any new data point.

Selecting meaningful features

If we notice that a model performs much better on a training dataset than on the test dataset, this observation is a strong indicator for **overfitting**. Overfitting means that model fits the parameters too closely to the particular observations in the training dataset but does not generalize well to real data—we say that the model has a *high variance*. A reason for overfitting is that our model is too complex for the given training data and common solutions to reduce the generalization error are listed as follows:

- Collect more training data
- Introduce a penalty for complexity via regularization
- Choose a simpler model with fewer parameters
- Reduce the dimensionality of the data

Collecting more training data is often not applicable. In the next chapter, we will learn about a useful technique to check whether more training data is helpful at all. In the following sections and subsections, we will look at common ways to reduce overfitting by regularization and dimensionality reduction via feature selection.

Sparse solutions with L1 regularization

We recall from *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, that **L2 regularization** is one approach to reduce the complexity of a model by penalizing large individual weights, where we defined the L2 norm of our weight vector *w* as follows:
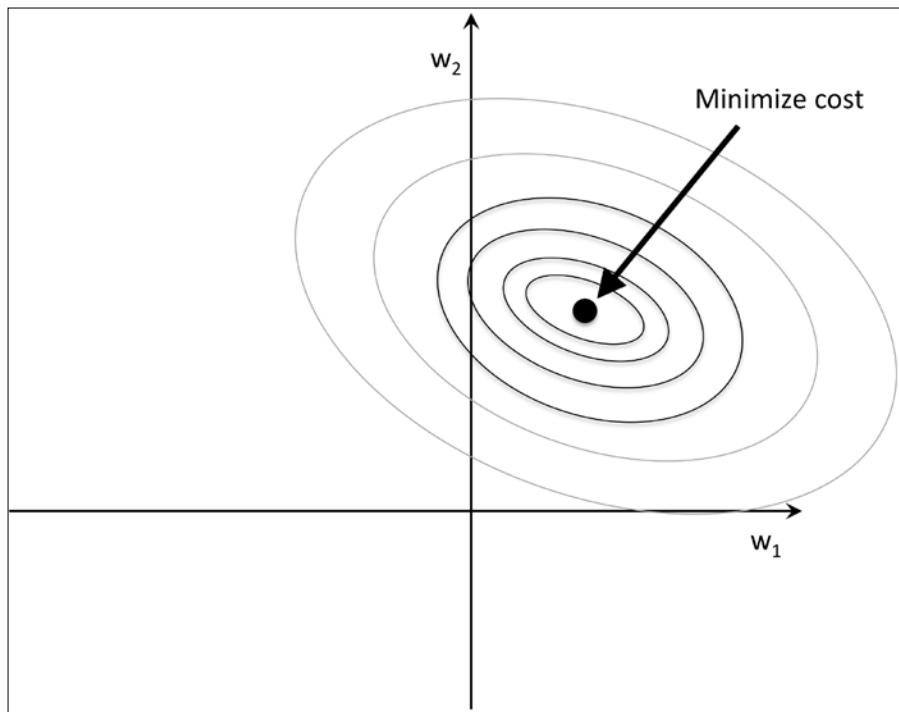
$$L2 : \left\| \boldsymbol{w} \right\|_2^2 = \sum_{j=1}^{m} w_j^2$$

Another approach to reduce the model complexity is the related **L1 regularization**:

$$L1 : \left\| \boldsymbol{w} \right\|_1 = \sum_{j=1}^{m} \left| w_j \right|$$
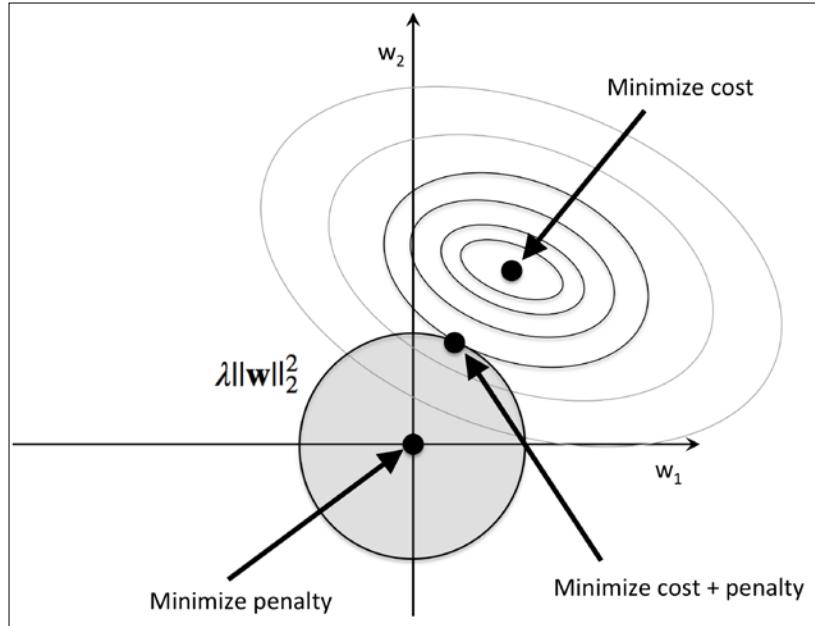
Here, we simply replaced the square of the weights by the sum of the absolute values of the weights. In contrast to L2 regularization, L1 regularization yields sparse feature vectors; most feature weights will be zero. Sparsity can be useful in practice if we have a high-dimensional dataset with many features that are irrelevant, especially cases where we have more irrelevant dimensions than samples. In this sense, L1 regularization can be understood as a technique for feature selection.

To better understand how L1 regularization encourages sparsity, let's take a step back and take a look at a geometrical interpretation of regularization. Let's plot the contours of a convex cost function for two weight coefficients w_1 and w_2. Here, we will consider the **sum of the squared errors** (**SSE**) cost function that we used for Adaline in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, since it is symmetrical and easier to draw than the cost function of logistic regression; however, the same concepts apply to the latter. Remember that our goal is to find the combination of weight coefficients that minimize the cost function for the training data, as shown in the following figure (the point in the middle of the ellipses):
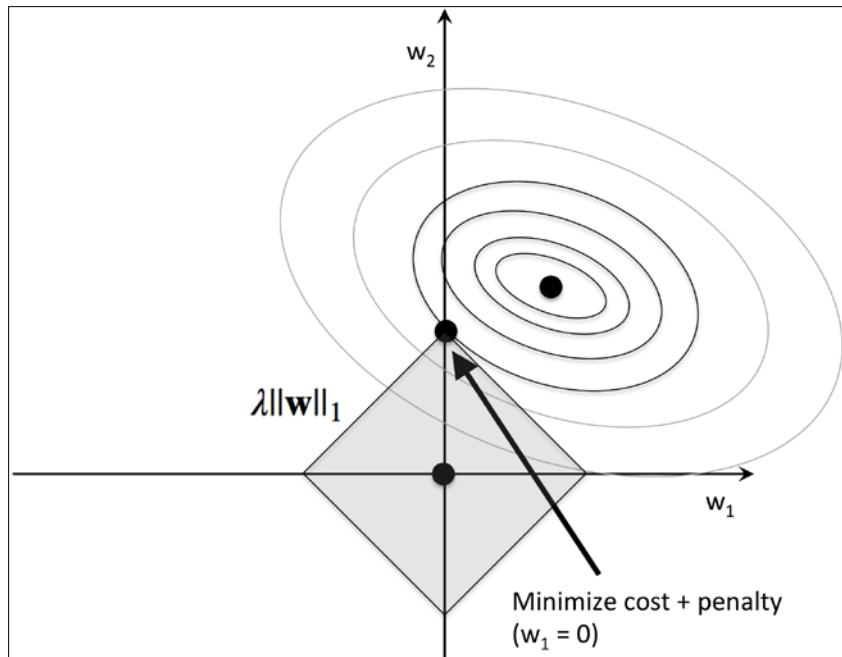
Now, we can think of regularization as adding a penalty term to the cost function to encourage smaller weights; or, in other words, we penalize large weights.

Thus, by increasing the regularization strength via the regularization parameter λ, we shrink the weights towards zero and decrease the dependence of our model on the training data. Let's illustrate this concept in the following figure for the L2 penalty term.

The quadratic L2 regularization term is represented by the shaded ball. Here, our weight coefficients cannot exceed our regularization *budget*—the combination of the weight coefficients cannot fall outside the shaded area. On the other hand, we still want to minimize the cost function. Under the penalty constraint, our best effort is to choose the point where the L2 ball intersects with the contours of the unpenalized cost function. The larger the value of the regularization parameter λ gets, the faster the penalized cost function grows, which leads to a narrower L2 ball. For example, if we increase the regularization parameter towards infinity, the weight coefficients will become effectively zero, denoted by the center of the L2 ball. To summarize the main message of the example: our goal is to minimize the sum of the unpenalized cost function plus the penalty term, which can be understood as adding bias and preferring a simpler model to reduce the variance in the absence of sufficient training data to fit the model.

Now let's discuss L1 regularization and sparsity. The main concept behind L1 regularization is similar to what we have discussed here. However, since the L1 penalty is the sum of the absolute weight coefficients (remember that the L2 term is quadratic), we can represent it as a diamond shape *budget*, as shown in the following figure:

In the preceding figure, we can see that the contour of the cost function touches the L1 diamond at $w_1 = 0$. Since the contours of an L1 regularized system are sharp, it is more likely that the optimum—that is, the intersection between the ellipses of the cost function and the boundary of the L1 diamond—is located on the axes, which encourages sparsity. The mathematical details of why L1 regularization can lead to sparse solutions are beyond the scope of this book. If you are interested, an excellent section on L2 versus L1 regularization can be found in section 3.4 of *The Elements of Statistical Learning, Trevor Hastie, Robert Tibshirani, and Jerome Friedman, Springer*.

For regularized models in scikit-learn that support L1 regularization, we can simply set the `penalty` parameter to `'l1'` to yield the sparse solution:

```
>>> from sklearn.linear_model import LogisticRegression
>>> LogisticRegression(penalty='l1')
```

Applied to the standardized Wine data, the L1 regularized logistic regression would yield the following sparse solution:

```
>>> lr = LogisticRegression(penalty='l1', C=0.1)
>>> lr.fit(X_train_std, y_train)
>>> print('Training accuracy:', lr.score(X_train_std, y_train))
Training accuracy: 0.983870967742
>>> print('Test accuracy:', lr.score(X_test_std, y_test))
Test accuracy: 0.981481481481
```

Both training and test accuracies (both 98 percent) do not indicate any overfitting of our model. When we access the intercept terms via the `lr.intercept_` attribute, we can see that the array returns three values:

```
>>> lr.intercept_
array([-0.38379237, -0.1580855 , -0.70047966])
```

Since we we the fit the `LogisticRegression` object on a multiclass dataset, it uses the **One-vs-Rest (OvR)** approach by default where the first intercept belongs to the model that fits class 1 versus class 2 and 3; the second value is the intercept of the model that fits class 2 versus class 1 and 3; and the third value is the intercept of the model that fits class 3 versus class 1 and 2, respectively:

```
>>> lr.coef_
array([[ 0.280, 0.000, 0.000, -0.0282, 0.000,
         0.000, 0.710, 0.000, 0.000, 0.000,
         0.000, 0.000, 1.236],
       [-0.644, -0.0688 , -0.0572, 0.000, 0.000,
         0.000, 0.000, 0.000, 0.000, -0.927,
         0.060, 0.000, -0.371],
       [ 0.000, 0.061, 0.000, 0.000, 0.000,
         0.000, -0.637, 0.000, 0.000, 0.499,
        -0.358, -0.570, 0.000
       ]])
```

The weight array that we accessed via the `lr.coef_` attribute contains three rows of weight coefficients, one weight vector for each class. Each row consists of 13 weights where each weight is multiplied by the respective feature in the 13-dimensional Wine dataset to calculate the net input:
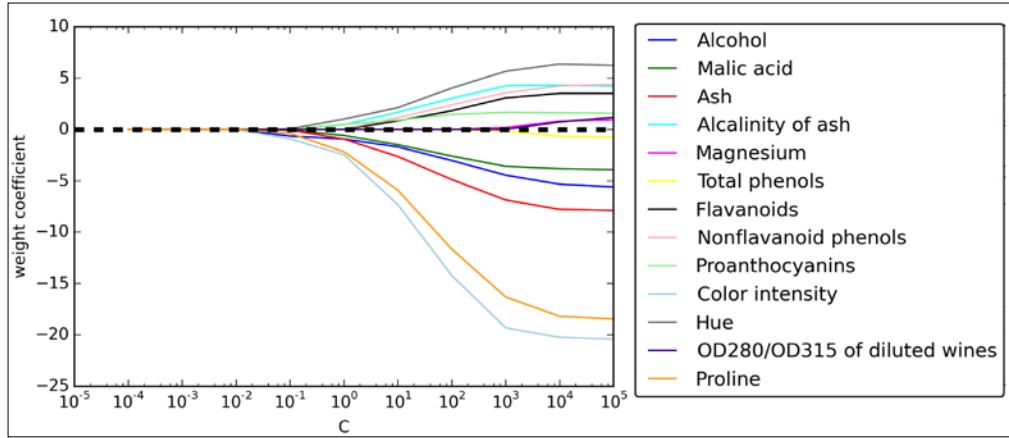
$$z = w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}$$

We notice that the weight vectors are sparse, which means that they only have a few non-zero entries. As a result of the L1 regularization, which serves as a method for feature selection, we just trained a model that is robust to the potentially irrelevant features in this dataset.

Lastly, let's plot the regularization path, which is the weight coefficients of the different features for different regularization strengths:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> colors = ['blue', 'green', 'red', 'cyan',
...           'magenta', 'yellow', 'black',
...           'pink', 'lightgreen', 'lightblue',
...           'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4, 6):
...     lr = LogisticRegression(penalty='l1',
...                             C=10**c,
...                             random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> for column, color in zip(range(weights.shape[1]), colors):
...     plt.plot(params, weights[:, column],
...              label=df_wine.columns[column+1],
...              color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
>>> ax.legend(loc='upper center',
...           bbox_to_anchor=(1.38, 1.03),
...           ncol=1, fancybox=True)
>>> plt.show()
```

The resulting plot provides us with further insights about the behavior of L1 regularization. As we can see, all features weights will be zero if we penalize the model with a strong regularization parameter ($C < 0.1$); C is the inverse of the regularization parameter λ.

Sequential feature selection algorithms

An alternative way to reduce the complexity of the model and avoid overfitting is **dimensionality reduction** via feature selection, which is especially useful for unregularized models. There are two main categories of dimensionality reduction techniques: **feature selection** and **feature extraction**. Using feature selection, we select a subset of the original features. In feature extraction, we derive information from the feature set to construct a new feature subspace. In this section, we will take a look at a classic family of feature selection algorithms. In the next chapter, *Chapter 5*, *Compressing Data via Dimensionality Reduction*, we will learn about different feature extraction techniques to compress a dataset onto a lower dimensional feature subspace.

Sequential feature selection algorithms are a family of greedy search algorithms that are used to reduce an initial *d-dimensional* feature space to a *k-dimensional* feature subspace where *k < d*. The motivation behind feature selection algorithms is to automatically select a subset of features that are most relevant to the problem to improve computational efficiency or reduce the generalization error of the model by removing irrelevant features or noise, which can be useful for algorithms that don't support regularization. A classic sequential feature selection algorithm is **Sequential Backward Selection** (**SBS**), which aims to reduce the dimensionality of the initial feature subspace with a minimum decay in performance of the classifier to improve upon computational efficiency. In certain cases, SBS can even improve the predictive power of the model if a model suffers from overfitting.

> Greedy algorithms make locally optimal choices at each stage of a combinatorial search problem and generally yield a suboptimal solution to the problem in contrast to exhaustive search algorithms, which evaluate all possible combinations and are guaranteed to find the optimal solution. However, in practice, an exhaustive search is often computationally not feasible, whereas greedy algorithms allow for a less complex, computationally more efficient solution.

The idea behind the SBS algorithm is quite simple: SBS sequentially removes features from the full feature subset until the new feature subspace contains the desired number of features. In order to determine which feature is to be removed at each stage, we need to define criterion function J that we want to minimize. The criterion calculated by the criterion function can simply be the difference in performance of the classifier after and before the removal of a particular feature. Then the feature to be removed at each stage can simply be defined as the feature that maximizes this criterion; or, in more intuitive terms, at each stage we eliminate the feature that causes the least performance loss after removal. Based on the preceding definition of SBS, we can outline the algorithm in 4 simple steps:

1. Initialize the algorithm with $k = d$, where d is the dimensionality of the full feature space $\mathbf{X}_d$.

2. Determine the feature x^- that maximizes the criterion $x^- = \arg\max J(X_k - x)$ where $x \in X_k$.

3. Remove the feature x^- from the feature set: $X_{k-1} := X_k - x^-; k := k - 1$.

4. Terminate if k equals the number of desired features, if not, go to step 2.

> You can find a detailed evaluation of several sequential feature algorithms in *Comparative Study of Techniques for Large Scale Feature Selection*, F. Ferri, P. Pudil, M. Hatef, and J. Kittler. *Comparative study of techniques for large-scale feature selection. Pattern Recognition in Practice IV, pages 403–413, 1994.*

Unfortunately, the SBS algorithm is not implemented in scikit-learn, yet. But since it is so simple, let's go ahead and implement it in Python from scratch:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.cross_validation import train_test_split
from sklearn.metrics import accuracy_score
```

```
class SBS():
    def __init__(self, estimator, k_features,
        scoring=accuracy_score,
        test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
                train_test_split(X, y, test_size=self.test_size,
                                    random_state=self.random_state)

        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                    X_test, y_test, self.indices_)
        self.scores_ = [score]

        while dim > self.k_features:
            scores = []
            subsets = []

            for p in combinations(self.indices_, r=dim-1):
                score = self._calc_score(X_train, y_train,
                                            X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]

        return self

    def transform(self, X):
        return X[:, self.indices_]
```

```
def _calc_score(self, X_train, y_train,
                        X_test, y_test, indices):
    self.estimator.fit(X_train[:, indices], y_train)
    y_pred = self.estimator.predict(X_test[:, indices])
    score = self.scoring(y_test, y_pred)
    return score
```

In the preceding implementation, we defined the `k_features` parameter to specify the desired number of features we want to return. By default, we use the `accuracy_score` from scikit-learn to evaluate the performance of a model and estimator for classification on the feature subsets. Inside the `while` loop of the `fit` method, the feature subsets created by the `itertools.combination` function are evaluated and reduced until the feature subset has the desired dimensionality. In each iteration, the accuracy score of the best subset is collected in a list `self.scores_` based on the internally created test dataset `X_test`. We will use those scores later to evaluate the results. The column indices of the final feature subset are assigned to `self.indices_`, which we can use via the `transform` method to return a new data array with the selected feature columns. Note that, instead of calculating the criterion explicitly inside the `fit` method, we simply removed the feature that is not contained in the best performing feature subset.

Now, let's see our SBS implementation in action using the KNN classifier from scikit-learn:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> import matplotlib.pyplot as plt
>>> knn = KNeighborsClassifier(n_neighbors=2)
>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```
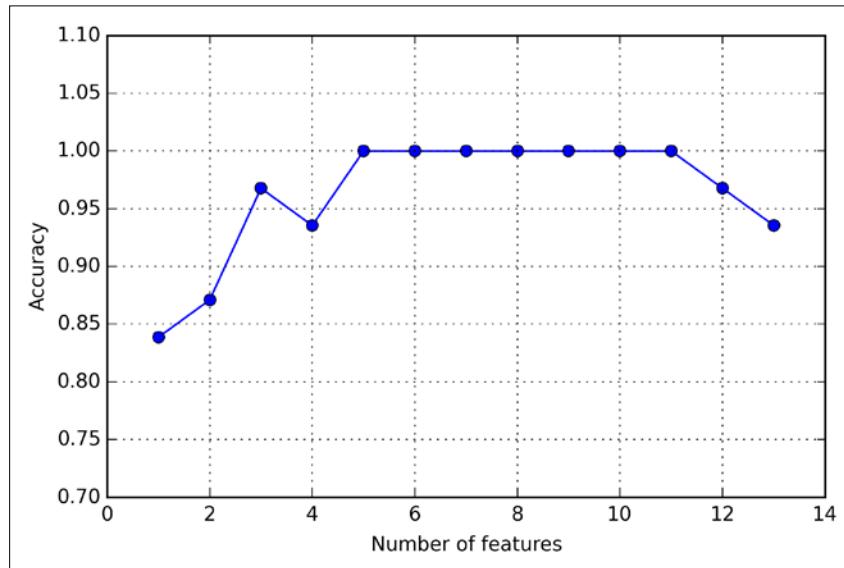
Although our SBS implementation already splits the dataset into a test and training dataset inside the `fit` function, we still fed the training dataset `X_train` to the algorithm. The SBS `fit` method will then create new training-subsets for testing (validation) and training, which is why this test set is also called **validation dataset**. *This approach is necessary to prevent our original test set becoming part of the training data.*

Remember that our SBS algorithm collects the scores of the best feature subset at each stage, so let's move on to the more exciting part of our implementation and plot the classification accuracy of the KNN classifier that was calculated on the validation dataset. The code is as follows:

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.1])
```

```
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Number of features')
>>> plt.grid()
>>> plt.show()
```

As we can see in the following plot, the accuracy of the KNN classifier improved on the validation dataset as we reduced the number of features, which is likely due to a decrease of **the curse of dimensionality** that we discussed in the context of the KNN algorithm in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Also, we can see in the following plot that the classifier achieved 100 percent accuracy for *k={5, 6, 7, 8, 9, 10}*:

To satisfy our own curiosity, let's see what those five features are that yielded such a good performance on the validation dataset:

```
>>> k5 = list(sbs.subsets_[8])
>>> print(df_wine.columns[1:][k5])
Index(['Alcohol', 'Malic acid', 'Alcalinity of ash', 'Hue',
'Proline'], dtype='object')
```

Using the preceding code, we obtained the column indices of the 5-feature subset from the 9th position in the `sbs.subsets_` attribute and returned the corresponding feature names from the column-index of the pandas Wine `DataFrame`.

Next let's evaluate the performance of the KNN classifier on the original test set:

```
>>> knn.fit(X_train_std, y_train)
>>> print('Training accuracy:', knn.score(X_train_std, y_train))
Training accuracy: 0.983870967742
>>> print('Test accuracy:', knn.score(X_test_std, y_test))
Test accuracy: 0.944444444444
```

In the preceding code, we used the complete feature set and obtained ~98.4 percent accuracy on the training dataset. However, the accuracy on the test dataset was slightly lower (~94.4 percent), which is an indicator of a slight degree of overfitting. Now let's use the selected 5-feature subset and see how well KNN performs:

```
>>> knn.fit(X_train_std[:, k5], y_train)
>>> print('Training accuracy:',
...          knn.score(X_train_std[:, k5], y_train))
Training accuracy: 0.959677419355
>>> print('Test accuracy:',
...          knn.score(X_test_std[:, k5], y_test))
Test accuracy: 0.962962962963
```

Using fewer than half of the original features in the Wine dataset, the prediction accuracy on the test set improved by almost 2 percent. Also, we reduced overfitting, which we can tell from the small gap between test (~96.3 percent) and training (~96.0 percent) accuracy.

> **Feature selection algorithms in scikit-learn**
>
> There are many more feature selection algorithms available via scikit-learn. Those include recursive backward elimination based on feature weights, tree-based methods to select features by importance, and univariate statistical tests. A comprehensive discussion of the different feature selection methods is beyond the scope of this book, but a good summary with illustrative examples can be found at http://scikit-learn.org/stable/modules/feature_selection.html.
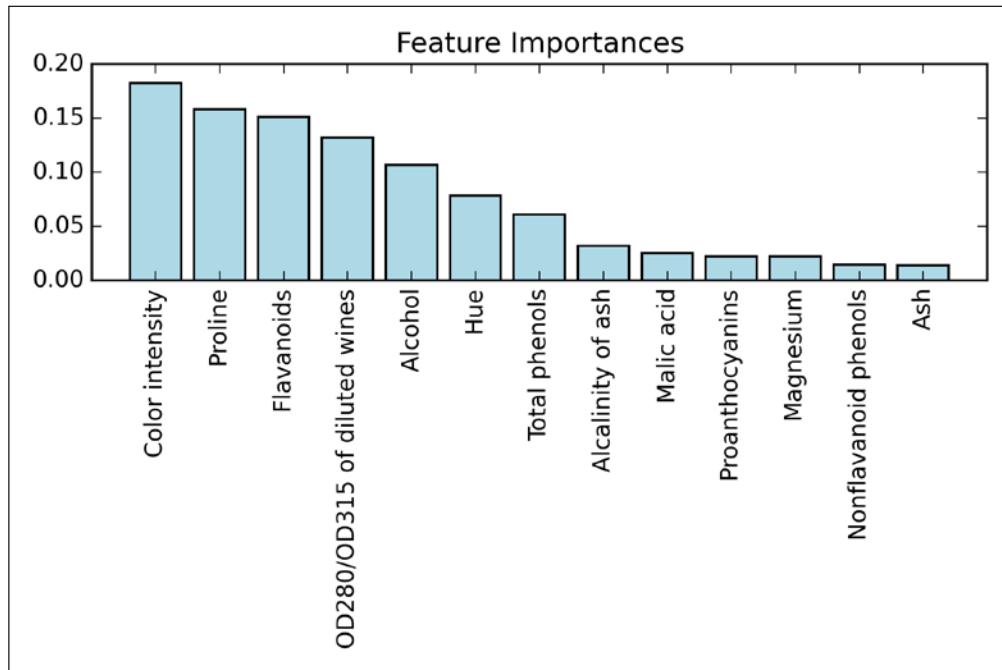
Assessing feature importance with random forests

In the previous sections, you learned how to use L1 regularization to zero out irrelevant features via logistic regression and use the SBS algorithm for feature selection. Another useful approach to select relevant features from a dataset is to use a random forest, an ensemble technique that we introduced in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*. Using a random forest, we can measure feature importance as the averaged impurity decrease computed from all decision trees in the forest without making any assumptions whether our data is linearly separable or not. Conveniently, the random forest implementation in scikit-learn already collects feature importances for us so that we can access them via the `feature_importances_` attribute after fitting a `RandomForestClassifier`. By executing the following code, we will now train a forest of 10,000 trees on the Wine dataset and rank the 13 features by their respective importance measures. Remember (from our discussion in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*) that we don't need to use standardized or normalized tree-based models. The code is as follows:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=10000,
...                                 random_state=0,
...                                 n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
...                             importances[indices[f]]))
 1) Color intensity                0.182483
 2) Proline                        0.158610
 3) Flavanoids                     0.150948
 4) OD280/OD315 of diluted wines   0.131987
 5) Alcohol                        0.106589
 6) Hue                            0.078243
 7) Total phenols                  0.060718
 8) Alcalinity of ash              0.032033
 9) Malic acid                     0.025400
10) Proanthocyanins                0.022351
11) Magnesium                      0.022078
```

```
12) Nonflavanoid phenols          0.014645
13) Ash                           0.013916
>>> plt.title('Feature Importances')
>>> plt.bar(range(X_train.shape[1]),
...         importances[indices],
...         color='lightblue',
...         align='center')
>>> plt.xticks(range(X_train.shape[1]),
...            feat_labels[indices], rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()
```

After executing the preceding code, we created a plot that ranks the different features in the Wine dataset by their relative importance; note that the feature importances are normalized so that they sum up to 1.0.

We can conclude that the color intensity of wine is the most discriminative feature in the dataset based on the average impurity decrease in the 10,000 decision trees. Interestingly, the three top-ranked features in the preceding plot are also among the top five features in the selection by the SBS algorithm that we implemented in the previous section. However, as far as interpretability is concerned, the random forest technique comes with an important *gotcha* that is worth mentioning. For instance, if two or more features are highly correlated, one feature may be ranked very highly while the information of the other feature(s) may not be fully captured. On the other hand, we don't need to be concerned about this problem if we are merely interested in the predictive performance of a model rather than the interpretation of feature importances. To conclude this section about feature importances and random forests, it is worth mentioning that scikit-learn also implements a `transform` method that selects features based on a user-specified threshold after model fitting, which is useful if we want to use the `RandomForestClassifier` as a feature selector and intermediate step in a scikit-learn pipeline, which allows us to connect different preprocessing steps with an estimator, as we will see in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*. For example, we could set the threshold to 0.15 to reduce the dataset to the 3 most important features, **Color intensity**, **Proline**, and **Flavonoids** using the following code:

```
>>> X_selected = forest.transform(X_train, threshold=0.15)
>>> X_selected.shape
(124, 3)
```

Summary

We started this chapter by looking at useful techniques to make sure that we handle missing data correctly. Before we feed data to a machine learning algorithm, we also have to make sure that we encode categorical variables correctly, and we have seen how we can map ordinal and nominal features values to integer representations.

Moreover, we briefly discussed L1 regularization, which can help us to avoid overfitting by reducing the complexity of a model. As an alternative approach for removing irrelevant features, we used a sequential feature selection algorithm to select meaningful features from a dataset.

In the next chapter, you will learn about yet another useful approach to dimensionality reduction: feature extraction. It allows us to compress features onto a lower dimensional subspace rather than removing features entirely as in feature selection.

5
Compressing Data via Dimensionality Reduction
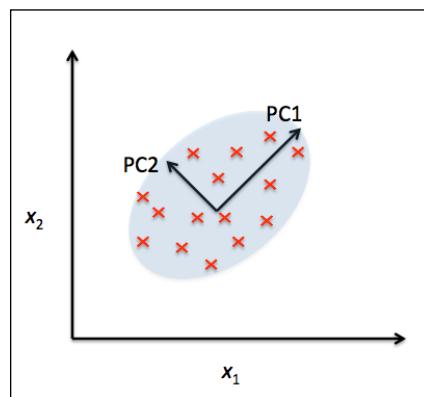
In *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, you learned about the different approaches for reducing the dimensionality of a dataset using different feature selection techniques. An alternative approach to feature selection for dimensionality reduction is *feature extraction*. In this chapter, you will learn about three fundamental techniques that will help us to summarize the information content of a dataset by transforming it onto a new feature subspace of lower dimensionality than the original one. Data compression is an important topic in machine learning, and it helps us to store and analyze the increasing amounts of data that are produced and collected in the modern age of technology. In this chapter, we will cover the following topics:

- **Principal component analysis** (**PCA**) for unsupervised data compression
- **Linear Discriminant Analysis** (**LDA**) as a supervised dimensionality reduction technique for maximizing class separability
- Nonlinear dimensionality reduction via **kernel principal component analysis**

Unsupervised dimensionality reduction via principal component analysis

Similar to feature selection, we can use feature extraction to reduce the number of features in a dataset. However, while we maintained the original features when we used feature selection algorithms, such as *sequential backward selection*, we use feature extraction to transform or project the data onto a new feature space. In the context of dimensionality reduction, feature extraction can be understood as an approach to data compression with the goal of maintaining most of the relevant information. Feature extraction is typically used to improve computational efficiency but can also help to reduce the *curse of dimensionality*—especially if we are working with nonregularized models.

Principal component analysis (**PCA**) is an unsupervised linear transformation technique that is widely used across different fields, most prominently for dimensionality reduction. Other popular applications of PCA include exploratory data analyses and de-noising of signals in stock market trading, and the analysis of genome data and gene expression levels in the field of bioinformatics. PCA helps us to identify patterns in data based on the correlation between features. In a nutshell, PCA aims to find the directions of maximum variance in high-dimensional data and projects it onto a new subspace with equal or fewer dimensions that the original one. The orthogonal axes (principal components) of the new subspace can be interpreted as the directions of maximum variance given the constraint that the new feature axes are orthogonal to each other as illustrated in the following figure. Here, x_1 and x_2 are the original feature axes, and **PC1** and **PC2** are the principal components:

If we use PCA for dimensionality reduction, we construct a $d \times k$-dimensional transformation matrix $\boldsymbol{W}$ that allows us to map a sample vector $\boldsymbol{x}$ onto a new k-dimensional feature subspace that has fewer dimensions than the original d-dimensional feature space:

$$\boldsymbol{x} = \left[x_1, x_2, \ldots, x_d \right], \quad \boldsymbol{x} \in \mathbb{R}^d$$

$$\downarrow \boldsymbol{xW}, \quad \boldsymbol{W} \in \mathbb{R}^{d \times k}$$

$$\mathbf{z} = \left[z_1, z_2, \ldots, z_k \right], \quad \mathbf{z} \in \mathbb{R}^k$$

As a result of transforming the original d-dimensional data onto this new k-dimensional subspace (typically $k \ll d$), the first principal component will have the largest possible variance, and all consequent principal components will have the largest possible variance given that they are uncorrelated (orthogonal) to the other principal components. Note that the PCA directions are highly sensitive to data scaling, and we need to standardize the features *prior* to PCA if the features were measured on different scales and we want to assign equal importance to all features.

Before looking at the PCA algorithm for dimensionality reduction in more detail, let's summarize the approach in a few simple steps:

1. Standardize the d-dimensional dataset.
2. Construct the covariance matrix.
3. Decompose the covariance matrix into its eigenvectors and eigenvalues.
4. Select k eigenvectors that correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \le d$).
5. Construct a projection matrix $\boldsymbol{W}$ from the "top" k eigenvectors.
6. Transform the d-dimensional input dataset X using the projection matrix $\boldsymbol{W}$ to obtain the new k-dimensional feature subspace.

Total and explained variance

In this subsection, we will tackle the first four steps of a principal component analysis: standardizing the data, constructing the covariance matrix, obtaining the eigenvalues and eigenvectors of the covariance matrix, and sorting the eigenvalues by decreasing order to rank the eigenvectors.

First, we will start by loading the *Wine* dataset that we have been working with in *Chapter 4, Building Good Training Sets – Data Preprocessing*:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
```

Next, we will process the *Wine* data into separate training and test sets—using 70 percent and 30 percent of the data, respectively—and standardize it to unit variance.

```
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...             train_test_split(X, y,
...             test_size=0.3, random_state=0)
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

After completing the mandatory preprocessing steps by executing the preceding code, let's advance to the second step: constructing the covariance matrix. The symmetric $d \times d$-dimensional covariance matrix, where d is the number of dimensions in the dataset, stores the pairwise covariances between the different features. For example, the covariance between two features x_j and x_k on the population level can be calculated via the following equation:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^{n} \left(x_j^{(i)} - \mu_j \right) \left(x_k^{(i)} - \mu_k \right)$$

Here, μ_j and μ_k are the sample means of feature j and k, respectively. Note that the sample means are zero if we standardize the dataset. A positive covariance between two features indicates that the features increase or decrease together, whereas a negative covariance indicates that the features vary in opposite directions. For example, a covariance matrix of three features can then be written as (note that Σ stands for the Greek letter *sigma*, which is not to be confused with the *sum* symbol):

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

The eigenvectors of the covariance matrix represent the principal components (the directions of maximum variance), whereas the corresponding eigenvalues will define their magnitude. In the case of the *Wine* dataset, we would obtain 13 eigenvectors and eigenvalues from the 13×13-dimensional covariance matrix.

Now, let's obtain the eigenpairs of the covariance matrix. As we surely remember from our introductory linear algebra or calculus classes, an eigen vector $\mathbf{v}$ satisfies the following condition:

$$\Sigma\mathbf{v} = \lambda\mathbf{v}$$

Here, λ is a scalar: the eigenvalue. Since the manual computation of eigenvectors and eigenvalues is a somewhat tedious and elaborate task, we will use the `linalg.eig` function from NumPy to obtain the eigenpairs of the *Wine* covariance matrix:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenvalues \n%s' % eigen_vals)
Eigenvalues
[ 4.8923083   2.46635032  1.42809973  1.01233462  0.84906459
0.60181514
0.52251546  0.08414846  0.33051429  0.29595018  0.16831254  0.21432212
0.2399553 ]
```

Using the `numpy.cov` function, we computed the covariance matrix of the standardized training dataset. Using the `linalg.eig` function, we performed the eigendecomposition that yielded a vector (`eigen_vals`) consisting of 13 eigenvalues and the corresponding eigenvectors stored as columns in a 13×13 -dimensional matrix (`eigen_vecs`).

> Although the `numpy.linalg.eig` function was designed to decompose nonsymmetric square matrices, you may find that it returns complex eigenvalues in certain cases.
>
> A related function, `numpy.linalg.eigh`, has been implemented to decompose Hermetian matrices, which is a numerically more stable approach to work with symmetric matrices such as the covariance matrix; `numpy.linalg.eigh` always returns real eigenvalues.

Since we want to reduce the dimensionality of our dataset by compressing it onto a new feature subspace, we only select the subset of the eigenvectors (principal components) that contains most of the information (variance). Since the eigenvalues define the magnitude of the eigenvectors, we have to sort the eigenvalues by decreasing magnitude; we are interested in the top k eigenvectors based on the values of their corresponding eigenvalues. But before we collect those k most informative eigenvectors, let's plot the *variance explained ratios* of the eigenvalues.

The variance explained ratio of an eigenvalue λ_j is simply the fraction of an eigenvalue λ_j and the total sum of the eigenvalues:
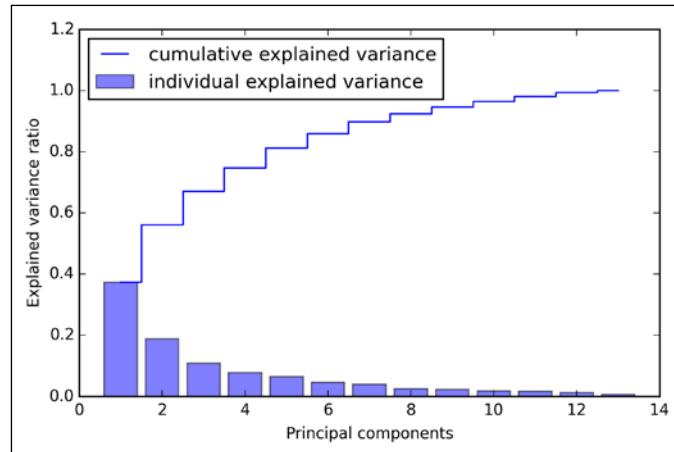
$$\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

Using the NumPy `cumsum` function, we can then calculate the cumulative sum of explained variances, which we will plot via matplotlib's `step` function:

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...            sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)

>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
...         label='individual explained variance')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...         label='cumulative explained variance')
>>> plt.ylabel('Explained variance ratio')
>>> plt.xlabel('Principal components')
>>> plt.legend(loc='best')
>>> plt.show()
```

The resulting plot indicates that the first principal component alone accounts for 40 percent of the variance. Also, we can see that the first two principal components combined explain almost 60 percent of the variance in the data:

Although the explained variance plot reminds us of the feature importance that we computed in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, via random forests, we shall remind ourselves that PCA is an unsupervised method, which means that information about the class labels is ignored. Whereas a random forest uses the class membership information to compute the node impurities, variance measures the spread of values along a feature axis.

Feature transformation

After we have successfully decomposed the covariance matrix into eigenpairs, let's now proceed with the last three steps to transform the *Wine* dataset onto the new principal component axes. In this section, we will sort the eigenpairs by descending order of the eigenvalues, construct a projection matrix from the selected eigenvectors, and use the projection matrix to transform the data onto the lower-dimensional subspace.

We start by sorting the eigenpairs by decreasing order of the eigenvalues:

```
>>> eigen_pairs =[(np.abs(eigen_vals[i]),eigen_vecs[:,i])
...                for i inrange(len(eigen_vals))]
>>> eigen_pairs.sort(reverse=True)
```

Next, we collect the two eigenvectors that correspond to the two largest values to capture about 60 percent of the variance in this dataset. Note that we only chose two eigenvectors for the purpose of illustration, since we are going to plot the data via a two-dimensional scatter plot later in this subsection. In practice, the number of principal components has to be determined from a trade-off between computational efficiency and the performance of the classifier:

```
>>> w= np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                eigen_pairs[1][1][:, np.newaxis]))
>>> print('Matrix W:\n',w)
Matrix W:
[[ 0.14669811  0.50417079]
[-0.24224554  0.24216889]
[-0.02993442  0.28698484]
[-0.25519002 -0.06468718]
[ 0.12079772  0.22995385]
[ 0.38934455  0.09363991]
[ 0.42326486  0.01088622]
[-0.30634956  0.01870216]
[ 0.30572219  0.03040352]
[-0.09869191  0.54527081]
```

```
  [ 0.30032535 -0.27924322]
  [ 0.36821154 -0.174365  ]
  [ 0.29259713  0.36315461]]
```

By executing the preceding code, we have created a 13×2-dimensional projection matrix W from the top two eigenvectors. Using the projection matrix, we can now transform a sample x (represented as 1×13-dimensional row vector) onto the PCA subspace obtaining x', a now two-dimensional sample vector consisting of two new features:

$$x' = xW$$

```
>>> X_train_std[0].dot(w)
array([ 2.59891628,  0.00484089])
```

Similarly, we can transform the entire 124×13-dimensional training dataset onto the two principal components by calculating the matrix dot product:
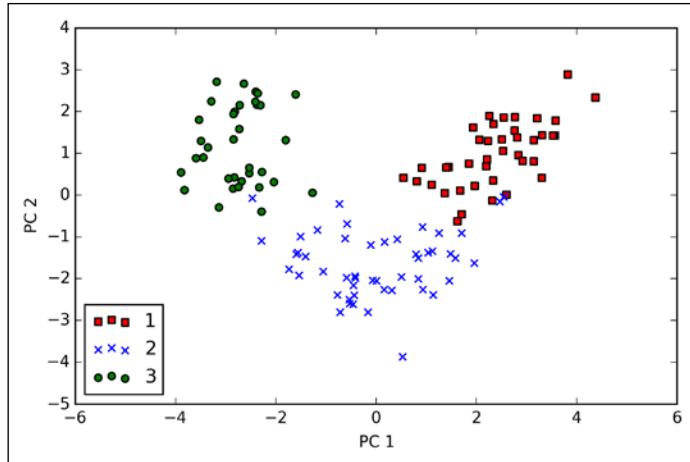
$$X' = XW$$

```
>>> X_train_pca = X_train_std.dot(w)
```

Lastly, let's visualize the transformed *Wine* training set, now stored as an 124×2-dimensional matrix, in a two-dimensional scatterplot:

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

As we can see in the resulting plot (shown in the next figure), the data is more spread along the *x*-axis—the first principal component—than the second principal component (*y*-axis), which is consistent with the explained variance ratio plot that we created in the previous subsection. However, we can intuitively see that a linear classifier will likely be able to separate the classes well:

Although we encoded the class labels information for the purpose of illustration in the preceding scatter plot, we have to keep in mind that PCA is an unsupervised technique that doesn't use class label information.

Principal component analysis in scikit-learn

Although the verbose approach in the previous subsection helped us to follow the inner workings of PCA, we will now discuss how to use the PCA class implemented in scikit-learn. PCA is another one of scikit-learn's transformer classes, where we first fit the model using the training data before we transform both the training data and the test data using the same model parameters. Now, let's use the PCA from scikit-learn on the *Wine* training dataset, classify the transformed samples via logistic regression, and visualize the decision regions via the plot_decision_region function that we defined in *Chapter 2, Training Machine Learning Algorithms for Classification*:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
```

```
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)

>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression()
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```
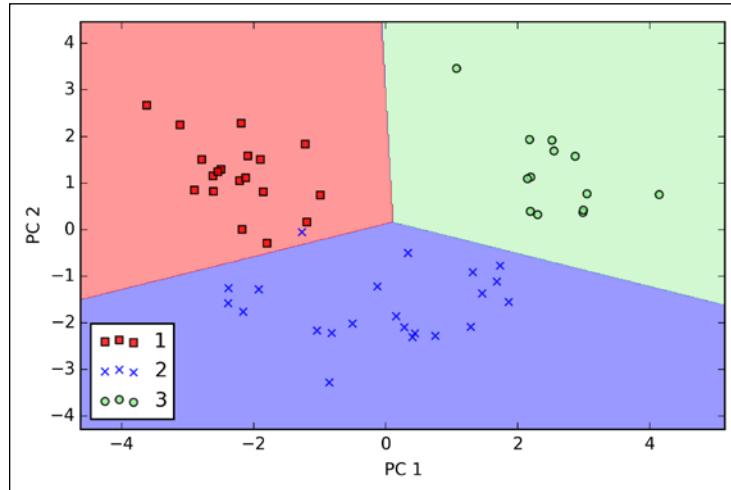
By executing the preceding code, we should now see the decision regions for the training model reduced to the two principal component axes.

If we compare the PCA projection via scikit-learn with our own PCA implementation, we notice that the plot above is a mirror image of the previous PCA via our step-by-step approach. Note that this is not due to an error in any of those two implementations, but the reason for this difference is that, depending on the eigensolver, eigenvectors can have either negative or positive signs. Not that it matters, but we could simply revert the mirror image by multiplying the data with `-1` if we wanted to; note that eigenvectors are typically scaled to unit length 1. For the sake of completeness, let's plot the decision regions of the logistic regression on the transformed test dataset to see if it can separate the classes well:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

After we plot the decision regions for the test set by executing the preceding code, we can see that logistic regression performs quite well on this small two-dimensional feature subspace and only misclassifies one sample in the test dataset.

If we are interested in the explained variance ratios of the different principal components, we can simply initialize the `PCA` class with the `n_components` parameter set to `None`, so all principal components are kept and the explained variance ratio can then be accessed via the `explained_variance_ratio_` attribute:

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
```

```
array([ 0.37329648,  0.18818926,  0.10896791,  0.07724389,
0.06478595,
0.04592014,  0.03986936,  0.02521914,  0.02258181,  0.01830924,
0.01635336,  0.01284271,  0.00642076])
```

Note that we set `n_components=None` when we initialized the PCA class so that it would return all principal components in sorted order instead of performing a dimensionality reduction.

Supervised data compression via linear discriminant analysis
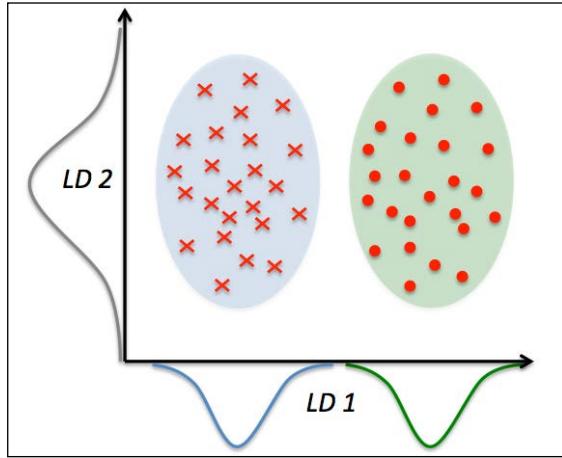
Linear Discriminant Analysis (**LDA**) can be used as a technique for feature extraction to increase the computational efficiency and reduce the degree of over-fitting due to the curse of dimensionality in nonregularized models.

The general concept behind LDA is very similar to PCA, whereas PCA attempts to find the orthogonal component axes of maximum variance in a dataset; the goal in LDA is to find the feature subspace that optimizes class separability. Both LDA and PCA are linear transformation techniques that can be used to reduce the number of dimensions in a dataset; the former is an unsupervised algorithm, whereas the latter is supervised. Thus, we might intuitively think that LDA is a superior feature extraction technique for classification tasks compared to PCA. However, A.M. Martinez reported that preprocessing via PCA tends to result in better classification results in an image recognition task in certain cases, for instance, if each class consists of only a small number of samples (A. M. Martinez and A. C. Kak. *PCA Versus LDA*. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 23(2):228–233, 2001).

> Although LDA is sometimes also called Fisher's LDA, Ronald A. Fisher initially formulated *Fisher's Linear Discriminant* for two-class classification problems in 1936 (R. A. Fisher. *The Use of Multiple Measurements in Taxonomic Problems*. Annals of Eugenics, 7(2):179–188, 1936). Fisher's Linear Discriminant was later generalized for multi-class problems by C. Radhakrishna Rao under the assumption of equal class covariances and normally distributed classes in 1948, which we now call LDA (C. R. Rao. *The Utilization of Multiple Measurements in Problems of Biological Classification*. Journal of the Royal Statistical Society. Series B (Methodological), 10(2):159–203, 1948).

The following figure summarizes the concept of LDA for a two-class problem. Samples from class 1 are shown as crosses and samples from class 2 are shown as circles, respectively:

A linear discriminant, as shown on the *x*-axis (LD 1), would separate the two normally distributed classes well. Although the exemplary linear discriminant shown on the *y*-axis (LD 2) captures a lot of the variance in the dataset, it would fail as a good linear discriminant since it does not capture any of the class-discriminatory information.

One assumption in LDA is that the data is normally distributed. Also, we assume that the classes have identical covariance matrices and that the features are statistically independent of each other. However, even if one or more of those assumptions are slightly violated, LDA for dimensionality reduction can still work reasonably well (R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. 2nd. Edition. New York, 2001).

Before we take a look into the inner workings of LDA in the following subsections, let's summarize the key steps of the LDA approach:

1. Standardize the d-dimensional dataset (d is the number of features).
2. For each class, compute the d-dimensional mean vector.
3. Construct the between-class scatter matrix S_B and the within-class scatter matrix S_w.

4. Compute the eigenvectors and corresponding eigenvalues of the matrix $S_w^{-1}S_B$.

5. Choose the k eigenvectors that correspond to the k largest eigenvalues to construct a $d \times k$-dimensional transformation matrix W; the eigenvectors are the columns of this matrix.

6. Project the samples onto the new feature subspace using the transformation matrix W.

> The assumptions that we make when we are using LDA are that the features are normally distributed and independent of each other. Also, the LDA algorithm assumes that the covariance matrices for the individual classes are identical. However, even if we violate those assumptions to a certain extent, LDA may still work reasonably well in dimensionality reduction and classification tasks (R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. 2nd. Edition. New York, 2001).

Computing the scatter matrices

Since we have already standardized the features of the *Wine* dataset in the PCA section at the beginning of this chapter, we can skip the first step and proceed with the calculation of the mean vectors, which we will use to construct the within-class scatter matrix and between-class scatter matrix, respectively. Each mean vector m_i stores the mean feature value μ_m with respect to the samples of class i:

$$m_i = \frac{1}{n_i} \sum_{x \in D_i}^{c} x_m$$

This results in three mean vectors:

$$m_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix} \quad i \in \{1,2,3\}$$

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...                 X_train_std[y_train==label], axis=0))
...     print('MV %s: %s\n' %(label, mean_vecs[label-1]))
MV 1: [ 0.9259 -0.3091  0.2592 -0.7989  0.3039  0.9608  1.0515 -0.6306
0.5354
  0.2209  0.4855  0.798   1.2017]

MV 2: [-0.8727 -0.3854 -0.4437  0.2481 -0.2409 -0.1059  0.0187 -0.0164
0.1095
 -0.8796  0.4392  0.2776 -0.7016]

MV 3: [ 0.1637  0.8929  0.3249  0.5658 -0.01   -0.9499 -1.228   0.7436
-0.7652
  0.979  -1.1698 -1.3007 -0.3912]
```

Using the mean vectors, we can now compute the within-class scatter matrix S_w:

$$S_W = \sum_{i=1}^{c} S_i$$

This is calculated by summing up the individual scatter matrices s_i of each individual class i:

$$S_i = \sum_{x \in D_i}^{c} \left(x - m_i\right)\left(x - m_i\right)^T$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1,4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     for row in X_train[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row-mv).dot((row-mv).T)
...     S_W += class_scatter
>>> print('Within-class scatter matrix: %sx%s'
...       % (S_W.shape[0], S_W.shape[1]))
Within-class scatter matrix: 13x13
```

The assumption that we are making when we are computing the scatter matrices is that the class labels in the training set are uniformly distributed. However, if we print the number of class labels, we see that this assumption is violated:

```
>>> print('Class label distribution: %s'
...       % np.bincount(y_train)[1:])
Class label distribution: [40 49 35]
```

Thus, we want to scale the individual scatter matrices S_i before we sum them up as scatter matrix S_w. When we divide the scatter matrices by the number of class samples N_i, we can see that computing the scatter matrix is in fact the same as computing the covariance matrix Σ_i. The covariance matrix is a normalized version of the scatter matrix:

$$\Sigma_i = \frac{1}{N_i} S_W = \frac{1}{N_i} \sum_{x \in D_i}^{c} (x - m_i)(x - m_i)^T$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>>> print('Scaled within-class scatter matrix: %sx%s'
...       % (S_W.shape[0], S_W.shape[1]))
Scaled within-class scatter matrix: 13x13
```

After we have computed the scaled within-class scatter matrix (or covariance matrix), we can move on to the next step and compute the between-class scatter matrix S_B:

$$S_B = \sum_{i=1}^{c} N_i (m_i - m)(m_i - m)^T$$

Here, m is the overall mean that is computed, including samples from all classes.

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13 # number of features
>>> S_B = np.zeros((d, d))
>>> for i,mean_vec in enumerate(mean_vecs):
...     n = X_train[y_train==i+1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1)
...     mean_overall = mean_overall.reshape(d, 1)
    S_B += n * (mean_vec - mean_overall).dot(
```

```
...                     (mean_vec - mean_overall).T)
print('Between-class scatter matrix: %sx%s'
...      % (S_B.shape[0], S_B.shape[1]))
Between-class scatter matrix: 13x13
```

Selecting linear discriminants for the new feature subspace

The remaining steps of the LDA are similar to the steps of the PCA. However, instead of performing the eigendecomposition on the covariance matrix, we solve the generalized eigenvalue problem of the matrix $S_w^{-1}S_B$:

```
>>>eigen_vals, eigen_vecs =\
...np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

After we computed the eigenpairs, we can now sort the eigenvalues in descending order:

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...                key=lambda k: k[0], reverse=True)
>>> print('Eigenvalues in decreasing order:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])


Eigenvalues in decreasing order:

452.721581245
156.43636122
8.11327596465e-14
2.78687384543e-14
2.78687384543e-14
2.27622032758e-14
2.27622032758e-14
1.97162599817e-14
1.32484714652e-14
1.32484714652e-14
1.03791501611e-14
5.94140664834e-15
2.12636975748e-16
```
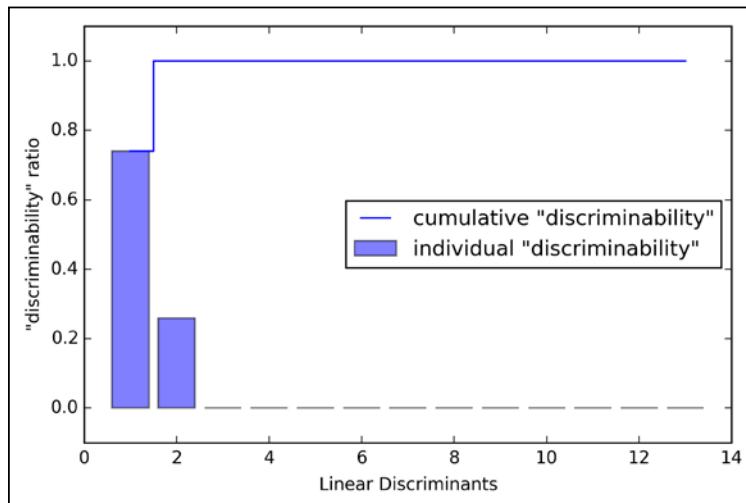
In LDA, the number of linear discriminants is at most $c-1$ where c is the number of class labels, since the in-between class scatter matrix S_B is the sum of c matrices with rank 1 or less. We can indeed see that we only have two nonzero eigenvalues (the eigenvalues 3-13 are not exactly zero, but this is due to the floating point arithmetic in NumPy). Note that in the rare case of perfect collinearity (all aligned sample points fall on a straight line), the covariance matrix would have rank one, which would result in only one eigenvector with a nonzero eigenvalue.

To measure how much of the class-discriminatory information is captured by the linear discriminants (eigenvectors), let's plot the linear discriminants by decreasing eigenvalues similar to the explained variance plot that we created in the PCA section. For simplicity, we will call the content of the class-discriminatory information *discriminability*.

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, alpha=0.5, align='center',
...          label='individual "discriminability"')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...          label='cumulative "discriminability"')
>>> plt.ylabel('"discriminability" ratio')
>>> plt.xlabel('Linear Discriminants')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.show()
```

As we can see in the resulting figure, the first two linear discriminants capture about 100 percent of the useful information in the *Wine* training dataset:

Let's now stack the two most discriminative eigenvector columns to create the transformation matrix W:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                   eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
[[ 0.0662 -0.3797]
[-0.0386 -0.2206]
[ 0.0217 -0.3816]
[-0.184 0.3018]
[ 0.0034 0.0141]
[-0.2326 0.0234]
[ 0.7747 0.1869]
[ 0.0811 0.0696]
[-0.0875 0.1796]
[-0.185 -0.284 ]
[ 0.066 0.2349]
[ 0.3805 0.073 ]
[ 0.3285 -0.5971]]
```
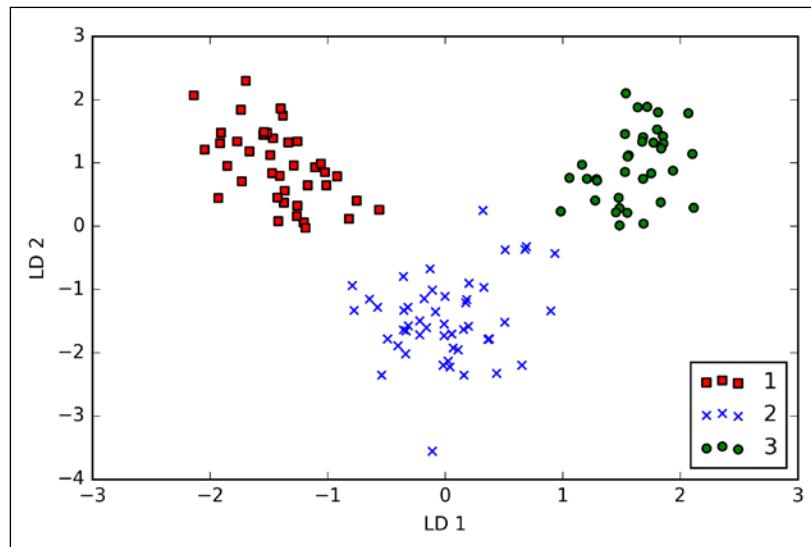
Projecting samples onto the new feature space

Using the transformation matrix W that we created in the previous subsection, we can now transform the training data set by multiplying the matrices:

$$X' = XW$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0]*(-1)
...                     X_train_lda[y_train==l, 1]*(-1)
...                     c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

As we can see in the resulting plot, the three wine classes are now linearly separable in the new feature subspace:

LDA via scikit-learn

The step-by-step implementation was a good exercise for understanding the inner workings of LDA and understanding the differences between LDA and PCA. Now, let's take a look at the `LDA` class implemented in scikit-learn:
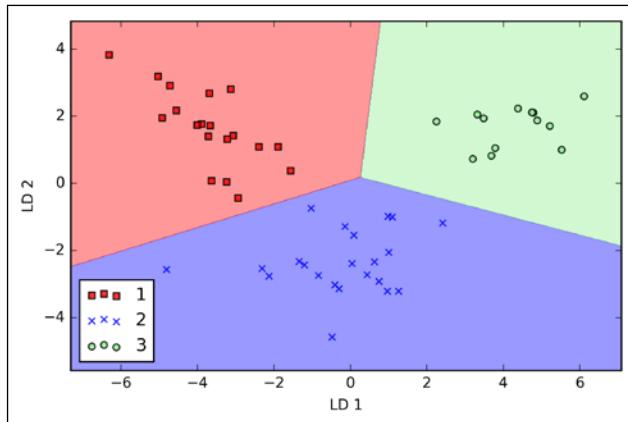
```
>>> from sklearn.lda import LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

Next, let's see how the logistic regression classifier handles the lower-dimensional training dataset after the LDA transformation:

```
>>> lr = LogisticRegression()
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```
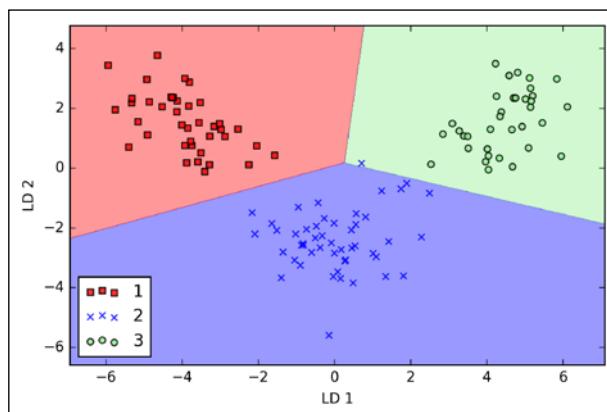
Looking at the resulting plot, we see that the logistic regression model misclassifies one of the samples from class 2:

By lowering the regularization strength, we could probably shift the decision boundaries so that the logistic regression models classify all samples in the training dataset correctly. However, let's take a look at the results on the test set:
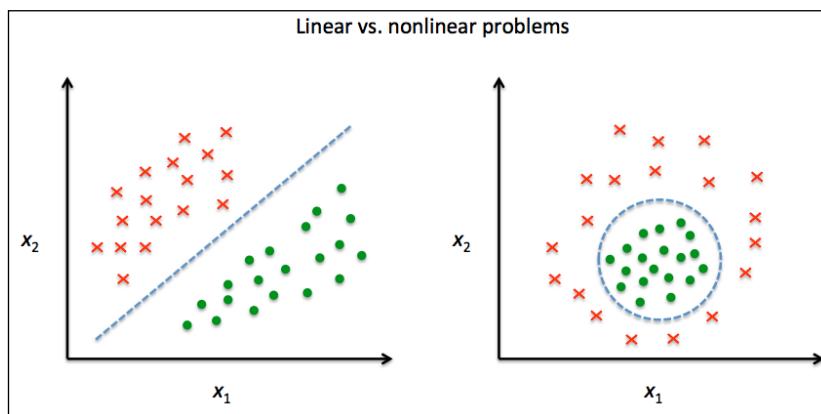
```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

As we can see in the resulting plot, the logistic regression classifier is able to get a perfect accuracy score for classifying the samples in the test dataset by only using a two-dimensional feature subspace instead of the original 13 *Wine* features:

Using kernel principal component analysis for nonlinear mappings

Many machine learning algorithms make assumptions about the linear separability of the input data. You learned that the perceptron even requires perfectly linearly separable training data to converge. Other algorithms that we have covered so far assume that the lack of perfect linear separability is due to noise: Adaline, logistic regression, and the (standard) **support vector machine** (**SVM**) to just name a few. However, if we are dealing with nonlinear problems, which we may encounter rather frequently in real-world applications, linear transformation techniques for dimensionality reduction, such as PCA and LDA, may not be the best choice. In this section, we will take a look at a kernelized version of PCA, or *kernel PCA*, which relates to the concepts of kernel SVM that we remember from *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Using kernel PCA, we will learn how to transform data that is not linearly separable onto a new, lower-dimensional subspace that is suitable for linear classifiers.

Kernel functions and the kernel trick

As we remember from our discussion about kernel SVMs in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we can tackle nonlinear problems by projecting them onto a new feature space of higher dimensionality where the classes become linearly separable. To transform the samples $x \in \mathbb{R}^d$ onto this higher k-dimensional subspace, we defined a nonlinear mapping function ϕ:

$$\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k \gg d)$$

We can think of ϕ as a function that creates nonlinear combinations of the original features to map the original d-dimensional dataset onto a larger, k-dimensional feature space. For example, if we had feature vector $\boldsymbol{x} \in \mathbb{R}^d$ ($\boldsymbol{x}$ is a column vector consisting of d features) with two dimensions ($d = 2$), a potential mapping onto a 3D space could be as follows:

$$\boldsymbol{x} = \left[x_1, \; x_2 \right]^T$$

$$\downarrow \phi$$

$$\mathbf{z} = \left[x_1^2, \sqrt{2 x_1 x_2}, x_2^2 \right]^T$$

In other words, via kernel PCA we perform a nonlinear mapping that transforms the data onto a higher-dimensional space and use standard PCA in this higher-dimensional space to project the data back onto a lower-dimensional space where the samples can be separated by a linear classifier (under the condition that the samples can be separated by density in the input space). However, one downside of this approach is that it is computationally very expensive, and this is where we use the *kernel trick*. Using the kernel trick, we can compute the similarity between two high-dimension feature vectors in the original feature space.

Before we proceed with more details about using the kernel trick to tackle this computationally expensive problem, let's look back at the *standard* PCA approach that we implemented at the beginning of this chapter. We computed the covariance between two features k and j as follows:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^{n} \left(x_j^{(i)} - \mu_j \right) \left(x_k^{(i)} - \mu_k \right)$$

Since the standardizing of features centers them at mean zero, for instance, $\mu_j = 0$ and $\mu_k = 0$, we can simplify this equation as follows:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^{n} x_j^{(i)} x_k^{(i)}$$

Note that the preceding equation refers to the covariance between two features; now, let's write the general equation to calculate the covariance *matrix* Σ:

$$\sum = \frac{1}{n}\sum_{i=1}^{n}x^{(i)}\,x^{(i)T}$$

Bernhard Scholkopf generalized this approach (B. Scholkopf, A. Smola, and K.-R. Muller. *Kernel Principal Component Analysis*. pages 583–588, 1997) so that we can replace the dot products between samples in the original feature space by the nonlinear feature combinations via ϕ:

$$\sum = \frac{1}{n}\sum_{i=1}^{n}\phi\left(x^{(i)}\right)\phi(x^{(i)})^{T}$$

To obtain the eigenvectors—the principal components—from this covariance matrix, we have to solve the following equation:

$$\Sigma v = \lambda v$$

$$\Rightarrow \frac{1}{n}\sum_{i=1}^{n}\phi\left(x^{(i)}\right)\phi\left(x^{(i)}\right)^{T}v = \lambda v$$

$$\Rightarrow v = \frac{1}{n\lambda}\sum_{i=1}^{n}\phi\left(x^{(i)}\right)\phi\left(x^{(i)}\right)^{T}v = \frac{1}{n}\sum_{i=1}^{n}a^{(i)}\phi\left(x^{(i)}\right)$$

Here, λ and v are the eigenvalues and eigenvectors of the covariance matrix Σ, and a can be obtained by extracting the eigenvectors of the kernel (similarity) matrix K as we will see in the following paragraphs.

The derivation of the kernel matrix is as follows:

First, let's write the covariance matrix as in matrix notation, where $\phi(X)$ is an $n \times k$ -dimensional matrix:

$$\sum = \frac{1}{n}\sum_{i=1}^{n}\phi\left(x^{(i)}\right)\phi\left(x^{(i)}\right)^{T} = \frac{1}{n}\phi(X)^{T}\phi(X)$$

Now, we can write the eigenvector equation as follows:

$$v = \frac{1}{n}\sum_{i=1}^{n} a^{(i)} \phi\left(x^{(i)}\right) = \lambda \phi(X)^T a$$

Since $\Sigma v = \lambda v$, we get:

$$\frac{1}{n}\phi(X)^T \phi(X)\phi(X)^T a = \lambda \phi(X)^T a$$

Multiplying it by $\phi(X)$ on both sides yields the following result:

$$\frac{1}{n}\phi(X)\phi(X)^T \phi(X)\phi(X)^T a = \lambda \phi(X)\phi(X)^T a$$

$$\Rightarrow \frac{1}{n}\phi(X)\phi(X)^T a = \lambda a$$

$$\Rightarrow \frac{1}{n}Ka = \lambda a$$

Here, K is the similarity (kernel) matrix:

$$K = \phi(X)\phi(X)^T$$

As we recall from the SVM section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we use the kernel trick to avoid calculating the pairwise dot products of the samples x under ϕ explicitly by using a kernel function K so that we don't need to calculate the eigenvectors explicitly:

$$k\left(x^{(i)}, x^{(j)}\right) = \phi\left(x^{(i)}\right)^T \phi\left(x^{(j)}\right)$$

In other words, what we obtain after kernel PCA are the samples already projected onto the respective components rather than constructing a transformation matrix as in the standard PCA approach. Basically, the kernel function (or simply *kernel*) can be understood as a function that calculates a dot product between two vectors — a measure of similarity.

The most commonly used kernels are as follows:

- The polynomial kernel:

$$k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \left(\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)} + \theta\right)^{p}$$

  Here, θ is the threshold and p is the power that has to be specified by the user.

- The hyperbolic tangent (sigmoid) kernel:

$$k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \tanh\left(\eta \boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)} + \theta\right)$$

- The **Radial Basis Function** (**RBF**) or Gaussian kernel that we will use in the following examples in the next subsection:

$$k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^{2}}{2\sigma^{2}}\right)$$

  It is also written as follows:

$$k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma \left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^{2}\right)$$

To summarize what we have discussed so far, we can define the following three steps to implement an RBF kernel PCA:

1. We compute the kernel (similarity) matrix k, where we need to calculate the following:

$$k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma \left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^{2}\right)$$

We do this for each pair of samples:

$$
K = \begin{bmatrix}
\kappa\left(x^{(1)}, x^{(1)}\right) & \kappa\left(x^{(1)}, x^{(2)}\right) & \cdots & \kappa\left(x^{(1)}, x^{(n)}\right) \\
\kappa\left(x^{(2)}, x^{(1)}\right) & \left(x^{(2)}, x^{(2)}\right) & \cdots & \kappa\left(x^{(2)}, x^{(n)}\right) \\
\vdots & \vdots & \ddots & \vdots \\
\kappa\left(x^{(n)}, x^{(1)}\right) & \kappa\left(x^{(n)}, x^{(2)}\right) & \cdots & \kappa\left(x^{(n)}, x^{(n)}\right)
\end{bmatrix}
$$

For example, if our dataset contains 100 training samples, the symmetric kernel matrix of the pair-wise similarities would be 100×100 dimensional.

2. We center the kernel matrix k using the following equation:

$$
K' = K - 1_n K - K 1_n + 1_n K 1_n
$$

Here, 1_n is an $n \times n$- dimensional matrix (the same dimensions as the kernel matrix) where all values are equal to $\frac{1}{n}$.

3. We collect the top k eigenvectors of the centered kernel matrix based on their corresponding eigenvalues, which are ranked by decreasing magnitude. In contrast to standard PCA, the eigenvectors are not the principal component axes but the samples projected onto those axes.

At this point, you may be wondering why we need to center the kernel matrix in the second step. We previously assumed that we are working with standardized data, where all features have mean zero when we formulated the covariance matrix and replaced the dot products by the nonlinear feature combinations via ϕ. Thus, the centering of the kernel matrix in the second step becomes necessary, since we do not compute the new feature space explicitly and we cannot guarantee that the new feature space is also centered at zero.

In the next section, we will put those three steps into action by implementing a kernel PCA in Python.

Implementing a kernel principal component analysis in Python

In the previous subsection, we discussed the core concepts behind kernel PCA. Now, we are going to implement an RBF kernel PCA in Python following the three steps that summarized the kernel PCA approach. Using the SciPy and NumPy helper functions, we will see that implementing a kernel PCA is actually really simple:

```python
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]

    gamma: float
      Tuning parameter of the RBF kernel

    n_components: int
      Number of principal components to return

    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)
```

```
# Center the kernel matrix.
N = K.shape[0]
one_n = np.ones((N,N)) / N
K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

# Obtaining eigenpairs from the centered kernel matrix
# numpy.eigh returns them in sorted order
eigvals, eigvecs = eigh(K)

# Collect the top k eigenvectors (projected samples)
X_pc = np.column_stack((eigvecs[:, -i]
                        for i in range(1, n_components + 1)))

return X_pc
```
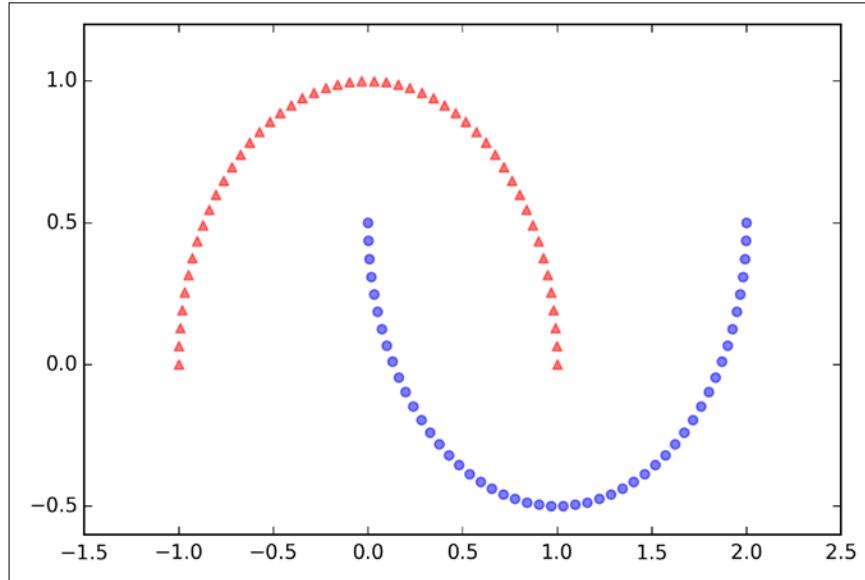
One downside of using an RBF kernel PCA for dimensionality reduction is that we have to specify the parameter γ a priori. Finding an appropriate value for γ requires experimentation and is best done using algorithms for parameter tuning, for example, grid search, which we will discuss in more detail in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Example 1 – separating half-moon shapes

Now, let's apply our `rbf_kernel_pca` on some nonlinear example datasets. We will start by creating a two-dimensional dataset of 100 sample points representing two half-moon shapes:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

For the purposes of illustration, the half-moon of triangular symbols shall represent one class and the half-moon depicted by the circular symbols represent the samples from another class:
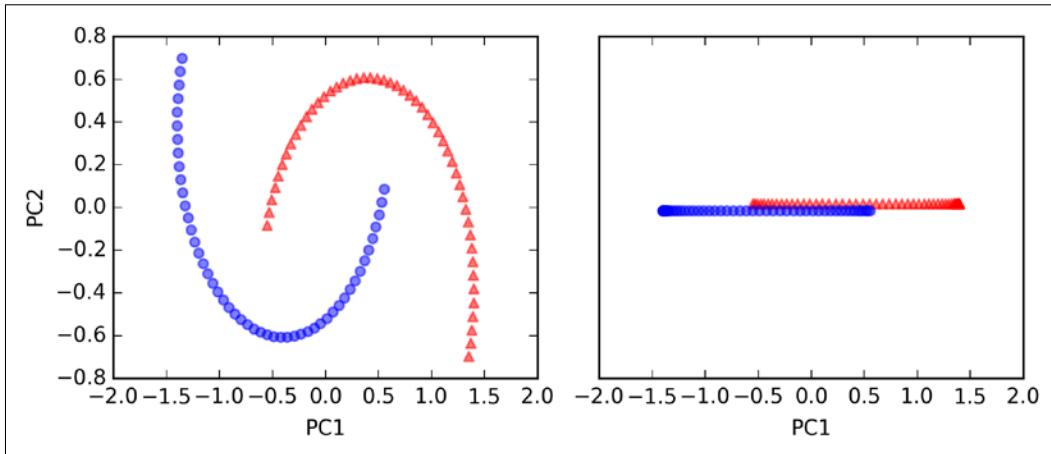
Clearly, these two half-moon shapes are not linearly separable and our goal is to *unfold* the half-moons via kernel PCA so that the dataset can serve as a suitable input for a linear classifier. But first, let's see what the dataset looks like if we project it onto the principal components via standard PCA:
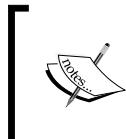
```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
```

```
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Clearly, we can see in the resulting figure that a linear classifier would be unable to perform well on the dataset transformed via standard PCA:

Note that when we plotted the first principal component only (right subplot), we shifted the triangular samples slightly upwards and the circular samples slightly downwards to better visualize the class overlap.
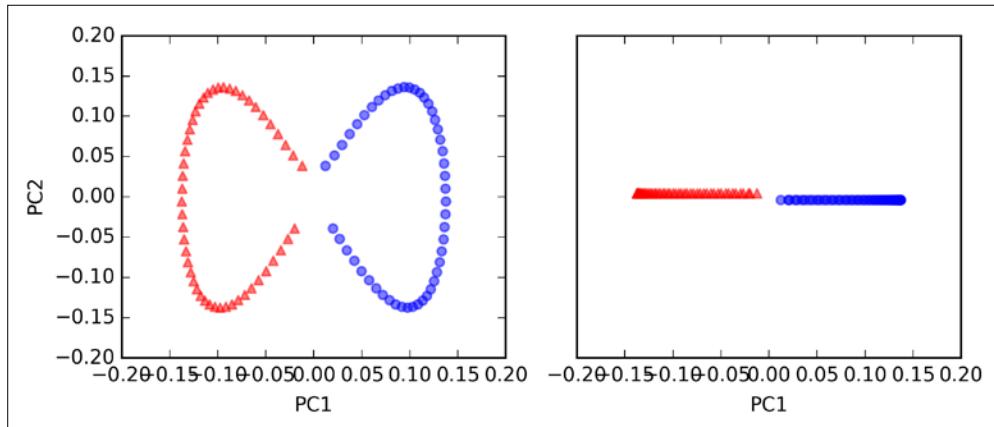
> Please remember that PCA is an unsupervised method and does not use class label information in order to maximize the variance in contrast to LDA. Here, the triangular and circular symbols were just added for visualization purposes to indicate the degree of separation.

Now, let's try out our kernel PCA function `rbf_kernel_pca`, which we implemented in the previous subsection:

```
>>> from matplotlib.ticker import FormatStrFormatter
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...                color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
```

```
...                    color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...                    color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...                    color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> plt.show()
```

We can now see that the two classes (circles and triangles) are linearly well separated so that it becomes a suitable training dataset for linear classifiers:

Unfortunately, there is no universal value for the tuning parameter γ that works well for different datasets. To find a γ value that is appropriate for a given problem requires experimentation. In *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will discuss techniques that can help us to automate the task of optimizing such tuning parameters. Here, I will use values for γ that I found produce *good* results.
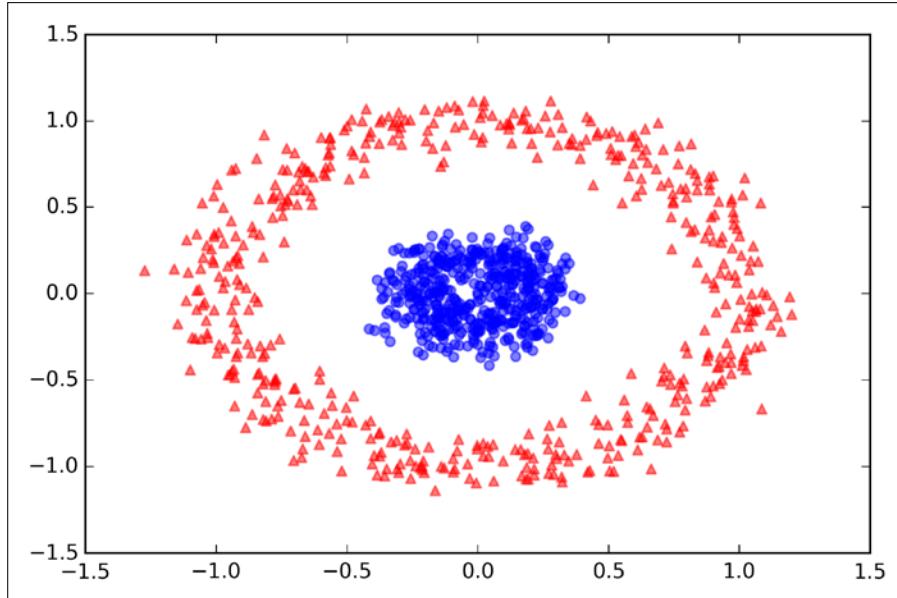
Example 2 – separating concentric circles

In the previous subsection, we showed you how to separate half-moon shapes via kernel PCA. Since we put so much effort into understanding the concepts of kernel PCA, let's take a look at another interesting example of a nonlinear problem: concentric circles.

The code is as follows:

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...             random_state=123, noise=0.1, factor=0.2)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.show()
```
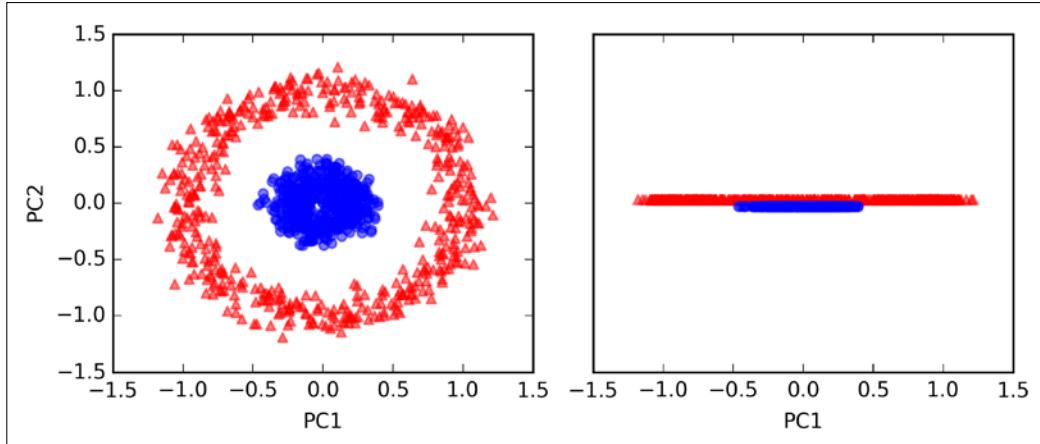
Again, we assume a two-class problem where the triangle shapes represent one class and the circle shapes represent another class, respectively:

Let's start with the standard PCA approach to compare it with the results of the RBF kernel PCA:

```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Again, we can see that standard PCA is not able to produce results suitable for training a linear classifier:

Given an appropriate value for γ, let's see if we are luckier using the RBF kernel PCA implementation:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...                color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...                color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...                color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...                color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Again, the RBF kernel PCA projected the data onto a new subspace where the two classes become linearly separable:

Projecting new data points

In the two previous example applications of kernel PCA, the half-moon shapes and the concentric circles, we projected a single dataset onto a new feature. In real applications, however, we may have more than one dataset that we want to transform, for example, training and test data, and typically also new samples we will collect after the model building and evaluation. In this section, you will learn how to project data points that were not part of the training dataset.

As we remember from the standard PCA approach at the beginning of this chapter, we project data by calculating the dot product between a transformation matrix and the input samples; the columns of the projection matrix are the top k eigenvectors (v) that we obtained from the covariance matrix. Now, the question is how can we transfer this concept to kernel PCA? If we think back to the idea behind kernel PCA, we remember that we obtained an eigenvector (a) of the centered kernel matrix (not the covariance matrix), which means that those are the samples that are already projected onto the principal component axis v. Thus, if we want to project a new sample x' onto this principal component axis, we'd need to compute the following:

$$\phi\left(x'\right)^{T} v$$

Fortunately, we can use the kernel trick so that we don't have to calculate the projection $\phi(x')^{T} v$ explicitly. However, it is worth noting that kernel PCA, in contrast to standard PCA, is a memory-based method, which means that we have to reuse the original training set each time to project new samples. We have to calculate the pairwise RBF kernel (similarity) between each ith sample in the training dataset and the new sample x':

$$\phi\left(x'\right)^{T} v = \sum_{i} a^{(i)} \phi\left(x'\right)^{T} \phi\left(x^{(i)}\right)$$

$$= \sum_{i} a^{(i)} k\left(x', x^{(i)}\right)^{T}$$

Here, eigenvectors a and eigenvalues λ of the Kernel matrix K satisfy the following condition in the equation:

$$Ka = \lambda a$$

After calculating the similarity between the new samples and the samples in the training set, we have to normalize the eigenvector a by its eigenvalue. Thus, let's modify the `rbf_kernel_pca` function that we implemented earlier so that it also returns the eigenvalues of the kernel matrix:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]

    gamma: float
      Tuning parameter of the RBF kernel

    n_components: int
      Number of principal components to return

    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset

     lambdas: list
       Eigenvalues

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)
```

```
# Center the kernel matrix.
N = K.shape[0]
one_n = np.ones((N,N)) / N
K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

# Obtaining eigenpairs from the centered kernel matrix
# numpy.eigh returns them in sorted order
eigvals, eigvecs = eigh(K)

# Collect the top k eigenvectors (projected samples)
alphas = np.column_stack((eigvecs[:,-i]
                for i in range(1,n_components+1)))

# Collect the corresponding eigenvalues
lambdas = [eigvals[-i] for i in range(1,n_components+1)]


return alphas, lambdas
```

Now, let's create a new half-moon dataset and project it onto a one-dimensional subspace using the updated RBF kernel PCA implementation:

```
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> alphas, lambdas =rbf_kernel_pca(X, gamma=15, n_components=1)
```

To make sure that we implement the code for projecting new samples, let's assume that the 26th point from the half-moon dataset is a new data point x', and our task is to project it onto this new subspace:

```
>>> x_new = X[25]
>>> x_new
array([ 1.8713187 ,  0.00928245])
>>> x_proj = alphas[25] # original projection
>>> x_proj
array([ 0.07877284])
>>> def project_x(x_new, X, gamma, alphas, lambdas):
...     pair_dist = np.array([np.sum(
...                    (x_new-row)**2) for row in X])
...     k = np.exp(-gamma * pair_dist)
... return k.dot(alphas / lambdas)
```
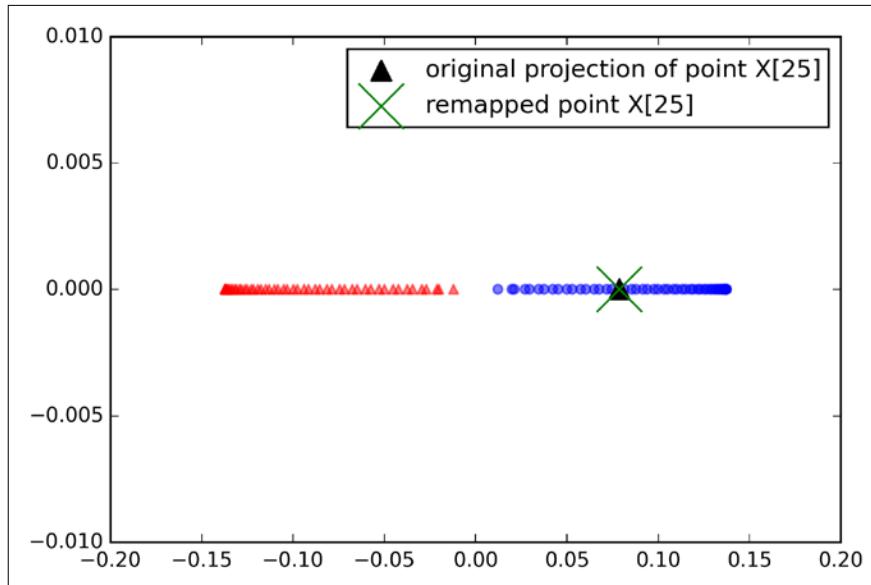
By executing the following code, we are able to reproduce the original projection. Using the `project_x` function, we will be able to project any new data samples as well. The code is as follows:

```
>>> x_reproj = project_x(x_new, X,
...         gamma=15, alphas=alphas, lambdas=lambdas)
>>> x_reproj
array([ 0.07877284])
```

Lastly, let's visualize the projection on the first principal component:

```
>>> plt.scatter(alphas[y==0, 0], np.zeros((50)),
...             color='red', marker='^',alpha=0.5)
>>> plt.scatter(alphas[y==1, 0], np.zeros((50)),
...             color='blue', marker='o', alpha=0.5)
>>> plt.scatter(x_proj, 0, color='black',
...             label='original projection of point X[25]',
...             marker='^', s=100)
>>> plt.scatter(x_reproj, 0, color='green',
...             label='remapped point X[25]',
...             marker='x', s=500)
>>> plt.legend(scatterpoints=1)
>>> plt.show()
```

As we can see in the following scatterplot, we mapped the sample x' onto the first principal component correctly:

Kernel principal component analysis in scikit-learn
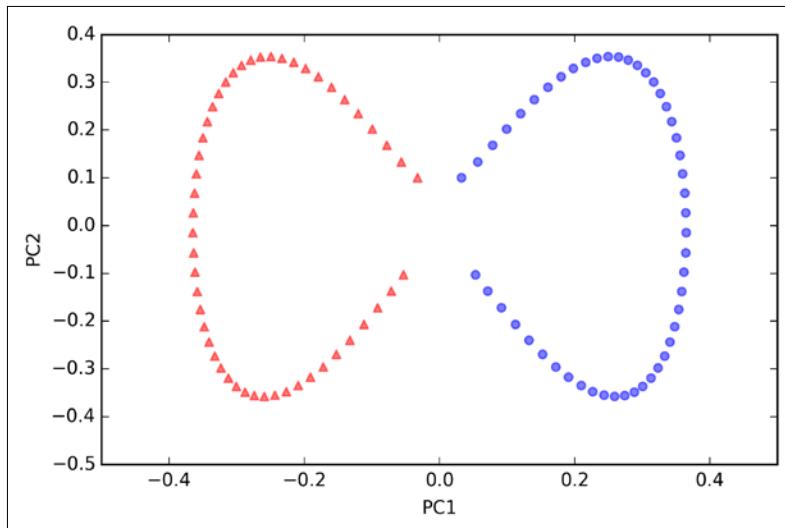
For our convenience, scikit-learn implements a kernel PCA class in the `sklearn.decomposition` submodule. The usage is similar to the standard PCA class, and we can specify the kernel via the `kernel` parameter:
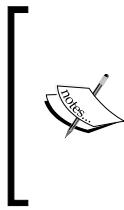
```
>>> from sklearn.decomposition import KernelPCA
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> scikit_kpca = KernelPCA(n_components=2,
...                 kernel='rbf', gamma=15)
>>> X_skernpca = scikit_kpca.fit_transform(X)
```

To see if we get results that are consistent with our own kernel PCA implementation, let's plot the transformed half-moon shape data onto the first two principal components:

```
>>> plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.show()
```

As we can see, the results of the scikit-learn `KernelPCA` are consistent with our own implementation:

Scikit-learn also implements advanced techniques for nonlinear dimensionality reduction that are beyond the scope of this book. You can find a nice overview of the current implementations in scikit-learn complemented with illustrative examples at `http://scikit-learn.org/stable/ modules/manifold.html`.

Summary

In this chapter, you learned about three different, fundamental dimensionality reduction techniques for feature extraction: standard PCA, LDA, and kernel PCA. Using PCA, we projected data onto a lower-dimensional subspace to maximize the variance along the orthogonal feature axes while ignoring the class labels. LDA, in contrast to PCA, is a technique for supervised dimensionality reduction, which means that it considers class information in the training dataset to attempt to maximize the class-separability in a linear feature space. Lastly, you learned about a kernelized version of PCA, which allows you to map nonlinear datasets onto a lower-dimensional feature space where the classes become linearly separable.

Equipped with these essential preprocessing techniques, you are now well prepared to learn about the best practices for efficiently incorporating different preprocessing techniques and evaluating the performance of different models in the next chapter.

6
Learning Best Practices for Model Evaluation and Hyperparameter Tuning

In the previous chapters, you learned about the essential machine learning algorithms for classification and how to get our data into shape before we feed it into those algorithms. Now, it's time to learn about the best practices of building good machine learning models by fine-tuning the algorithms and evaluating the model's performance! In this chapter, we will learn how to:

- Obtain unbiased estimates of a model's performance
- Diagnose the common problems of machine learning algorithms
- Fine-tune machine learning models
- Evaluate predictive models using different performance metrics

Streamlining workflows with pipelines

When we applied different preprocessing techniques in the previous chapters, such as **standardization** for feature scaling in *Chapter 4, Building Good Training Sets – Data Preprocessing*, or **principal component analysis** for data compression in *Chapter 5, Compressing Data via Dimensionality Reduction*, you learned that we have to reuse the parameters that were obtained during the fitting of the training data to scale and compress any new data, for example, the samples in the separate test dataset. In this section, you will learn about an extremely handy tool, the `Pipeline` class in scikit-learn. It allows us to fit a model including an arbitrary number of transformation steps and apply it to make predictions about new data.

Loading the Breast Cancer Wisconsin dataset

In this chapter, we will be working with the **Breast Cancer Wisconsin** dataset, which contains 569 samples of **malignant** and **benign** tumor cells. The first two columns in the dataset store the unique ID numbers of the samples and the corresponding diagnosis (*M=malignant*, *B=benign*), respectively. The columns 3-32 contain 30 real-value features that have been computed from digitized images of the cell nuclei, which can be used to build a model to predict whether a tumor is benign or malignant. The Breast Cancer Wisconsin dataset has been deposited on the *UCI machine learning repository* and more detailed information about this dataset can be found at `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`.

In this section we will read in the dataset, and split it into training and test datasets in three simple steps:

1. We will start by reading in the dataset directly from the UCI website using pandas:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/breast-cancer-wisconsin/wdbc.data',
header=None)
```

2. Next, we assign the 30 features to a NumPy array `X`. Using `LabelEncoder`, we transform the class labels from their original string representation (`M` and `B`) into integers:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2:].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

After encoding the class labels (diagnosis) in an array `y`, the malignant tumors are now represented as class `1`, and the benign tumors are represented as class `0`, respectively, which we can illustrate by calling the `transform` method of `LabelEncoder` on two dummy class labels:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

3. Before we construct our first model pipeline in the following subsection, let's divide the dataset into a separate training dataset (80 percent of the data) and a separate test dataset (20 percent of the data):

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = \
...        train_test_split(X, y, test_size=0.20, random_state=1)
```
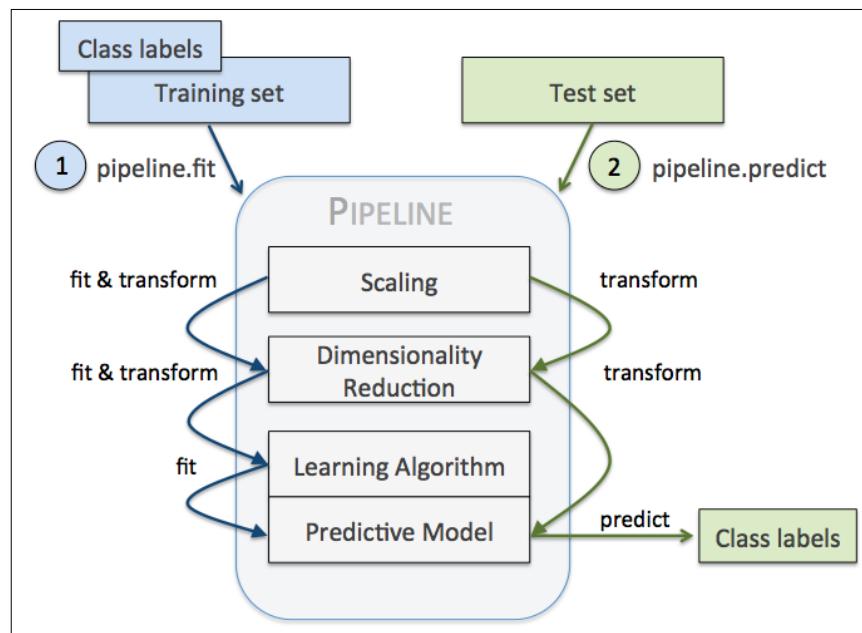
Combining transformers and estimators in a pipeline

In the previous chapter, you learned that many learning algorithms require input features on the same scale for optimal performance. Thus, we need to standardize the columns in the Breast Cancer Wisconsin dataset before we can feed them to a linear classifier, such as logistic regression. Furthermore, let's assume that we want to compress our data from the initial 30 dimensions onto a lower two-dimensional subspace via **principal component analysis** (**PCA**), a feature extraction technique for dimensionality reduction that we introduced in *Chapter 5*, *Compressing Data via Dimensionality Reduction*. Instead of going through the fitting and transformation steps for the training and test dataset separately, we can chain the StandardScaler, PCA, and LogisticRegression objects in a pipeline:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import Pipeline
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...               ('pca', PCA(n_components=2)),
...               ('clf', LogisticRegression(random_state=1))])
>>> pipe_lr.fit(X_train, y_train)
>>> print('Test Accuracy: %.3f' % pipe_lr.score(X_test, y_test))
Test Accuracy: 0.947
```

The Pipeline object takes a list of tuples as input, where the first value in each tuple is an arbitrary identifier string that we can use to access the individual elements in the pipeline, as we will see later in this chapter, and the second element in every tuple is a scikit-learn transformer or estimator.

The intermediate steps in a pipeline constitute scikit-learn transformers, and the last step is an estimator. In the preceding code example, we built a pipeline that consisted of two intermediate steps, a `StandardScaler` and a `PCA` transformer, and a logistic regression classifier as a final estimator. When we executed the `fit` method on the pipeline `pipe_lr`, the `StandardScaler` performed `fit` and `transform` on the training data, and the transformed training data was then passed onto the next object in the pipeline, the `PCA`. Similar to the previous step, `PCA` also executed `fit` and `transform` on the scaled input data and passed it to the final element of the pipeline, the estimator. We should note that there is no limit to the number of intermediate steps in this pipeline. The concept of how pipelines work is summarized in the following figure:
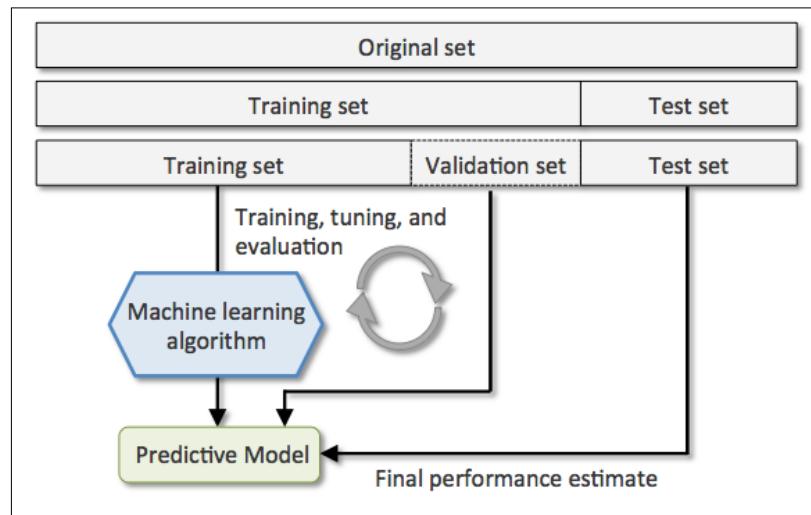
Using k-fold cross-validation to assess model performance

One of the key steps in building a machine learning model is to estimate its performance on data that the model hasn't seen before. Let's assume that we fit our model on a training dataset and use the same data to estimate how well it performs in practice. We remember from the *Tackling overfitting via regularization* section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, that a model can either suffer from underfitting (high bias) if the model is too simple, or it can overfit the training data (high variance) if the model is too complex for the underlying training data. To find an acceptable bias-variance trade-off, we need to evaluate our model carefully. In this section, you will learn about the useful cross-validation techniques **holdout cross-validation** and **k-fold cross-validation**, which can help us to obtain reliable estimates of the model's generalization error, that is, how well the model performs on unseen data.

The holdout method

A classic and popular approach for estimating the generalization performance of machine learning models is holdout cross-validation. Using the holdout method, we split our initial dataset into a separate training and test dataset — the former is used for model training, and the latter is used to estimate its performance. However, in typical machine learning applications, we are also interested in tuning and comparing different parameter settings to further improve the performance for making predictions on unseen data. This process is called **model selection**, where the term model selection refers to a given classification problem for which we want to select the *optimal* values of tuning parameters (also called **hyperparameters**). However, if we reuse the same test dataset over and over again during model selection, it will become part of our training data and thus the model will be more likely to overfit. Despite this issue, many people still use the test set for model selection, which is not a good machine learning practice.

A better way of using the holdout method for model selection is to separate the data into three parts: a training set, a validation set, and a test set. The training set is used to fit the different models, and the performance on the validation set is then used for the model selection. The advantage of having a test set that the model hasn't seen before during the training and model selection steps is that we can obtain a less biased estimate of its ability to generalize to new data. The following figure illustrates the concept of holdout cross-validation where we use a validation set to repeatedly evaluate the performance of the model after training using different parameter values. Once we are satisfied with the tuning of parameter values, we estimate the models' generalization error on the test dataset:

A disadvantage of the holdout method is that the performance estimate is sensitive to how we partition the training set into the training and validation subsets; the estimate will vary for different samples of the data. In the next subsection, we will take a look at a more robust technique for performance estimation, k-fold cross-validation, where we repeat the holdout method k times on k subsets of the training data.

K-fold cross-validation

In k-fold cross-validation, we randomly split the training dataset into *k* folds without replacement, where $k-1$ folds are used for the model training and one fold is used for testing. This procedure is repeated *k* times so that we obtain *k* models and performance estimates.
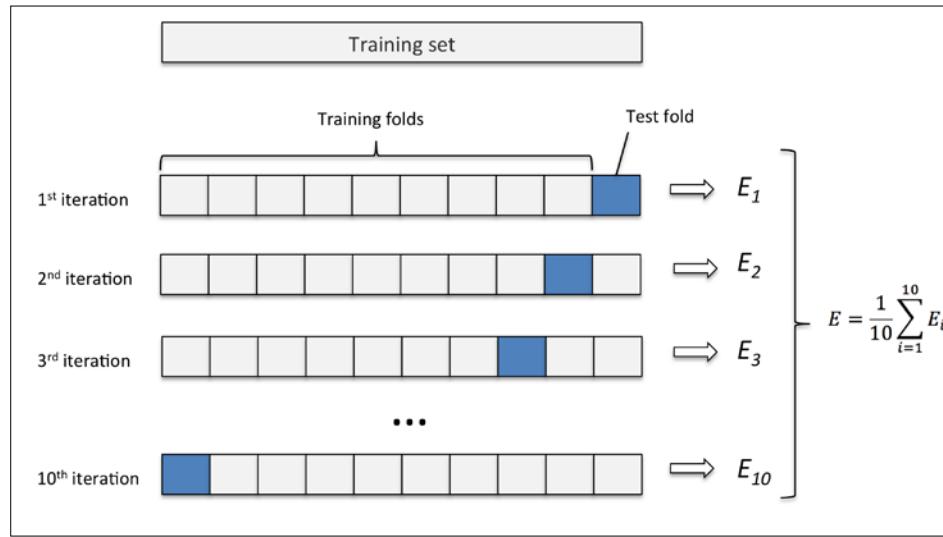
> In case you are not familiar with the terms sampling *with* and *without* replacement, let's walk through a simple thought experiment. Let's assume we are playing a lottery game where we randomly draw numbers from an urn. We start with an urn that holds five unique numbers 0, 1, 2, 3, and 4, and we draw exactly one number each turn. In the first round, the chance of drawing a particular number from the urn would be 1/5. Now, in sampling without replacement, we do not put the number back into the urn after each turn. Consequently, the probability of drawing a particular number from the set of remaining numbers in the next round depends on the previous round. For example, if we have a remaining set of numbers 0, 1, 2, and 4, the chance of drawing number 0 would become 1/4 in the next turn.
>
> However, in random sampling with replacement, we always return the drawn number to the urn so that the probabilities of drawing a particular number at each turn does not change; we can draw the same number more than once. In other words, in sampling with replacement, the samples (numbers) are independent and have a covariance zero. For example, the results from five rounds of drawing random numbers could look like this:
>
> - Random sampling without replacement: 2, 1, 3, 4, 0
> - Random sampling with replacement: 1, 3, 3, 4, 1

We then calculate the average performance of the models based on the different, independent folds to obtain a performance estimate that is less sensitive to the subpartitioning of the training data compared to the holdout method. Typically, we use k-fold cross-validation for model tuning, that is, finding the optimal hyperparameter values that yield a satisfying generalization performance. Once we have found satisfactory hyperparameter values, we can retrain the model on the complete training set and obtain a final performance estimate using the independent test set.

Since k-fold cross-validation is a resampling technique without replacement, the advantage of this approach is that each sample point will be part of a training and test dataset exactly once, which yields a lower-variance estimate of the model performance than the holdout method. The following figure summarizes the concept behind k-fold cross-validation with $k = 10$. The training data set is divided into 10 folds, and during the 10 iterations, 9 folds are used for training, and 1 fold will be used as the test set for the model evaluation. Also, the estimated performances E_i (for example, classification accuracy or error) for each fold are then used to calculate the estimated average performance E of the model:

The standard value for *k* in k-fold cross-validation is 10, which is typically a reasonable choice for most applications. However, if we are working with relatively small training sets, it can be useful to increase the number of folds. If we increase the value of *k*, more training data will be used in each iteration, which results in a lower bias towards estimating the generalization performance by averaging the individual model estimates. However, large values of *k* will also increase the runtime of the cross-validation algorithm and yield estimates with higher variance since the training folds will be more similar to each other. On the other hand, if we are working with large datasets, we can choose a smaller value for *k*, for example, $k = 5$, and still obtain an accurate estimate of the average performance of the model while reducing the computational cost of refitting and evaluating the model on the different folds.

> A special case of k-fold cross validation is the **leave-one-out** (**LOO**) cross-validation method. In LOO, we set the number of folds equal to the number of training samples (*k = n*) so that only one training sample is used for testing during each iteration. This is a recommended approach for working with very small datasets.

A slight improvement over the standard k-fold cross-validation approach is stratified k-fold cross-validation, which can yield better bias and variance estimates, especially in cases of unequal class proportions, as it has been shown in a study by R. Kohavi et al. (R. Kohavi et al. *A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection*. In Ijcai, volume 14, pages 1137–1145, 1995). In stratified cross-validation, the class proportions are preserved in each fold to ensure that each fold is representative of the class proportions in the training dataset, which we will illustrate by using the StratifiedKFold iterator in scikit-learn:

```
>>> import numpy as np
>>> from sklearn.cross_validation import StratifiedKFold
>>> kfold = StratifiedKFold(y=y_train,
...                         n_folds=10,
...                         random_state=1)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print('Fold: %s, Class dist.: %s, Acc: %.3f' % (k+1,
...                 np.bincount(y_train[train]), score))
Fold: 1, Class dist.: [256 153], Acc: 0.891
Fold: 2, Class dist.: [256 153], Acc: 0.978
Fold: 3, Class dist.: [256 153], Acc: 0.978
Fold: 4, Class dist.: [256 153], Acc: 0.913
Fold: 5, Class dist.: [256 153], Acc: 0.935
Fold: 6, Class dist.: [257 153], Acc: 0.978
Fold: 7, Class dist.: [257 153], Acc: 0.933
Fold: 8, Class dist.: [257 153], Acc: 0.956
Fold: 9, Class dist.: [257 153], Acc: 0.978
Fold: 10, Class dist.: [257 153], Acc: 0.956
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                 np.mean(scores), np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

First, we initialized the `StratifiedKfold` iterator from the `sklearn.cross_validation` module with the class labels `y_train` in the training set, and specified the number of folds via the `n_folds` parameter. When we used the `kfold` iterator to loop through the `k` folds, we used the returned indices in `train` to fit the logistic regression pipeline that we set up at the beginning of this chapter. Using the `pile_lr` pipeline, we ensured that the samples were scaled properly (for instance, standardized) in each iteration. We then used the `test` indices to calculate the accuracy score of the model, which we collected in the `scores` list to calculate the average accuracy and the standard deviation of the estimate.

Although the previous code example was useful to illustrate how k-fold cross-validation works, scikit-learn also implements a k-fold cross-validation scorer, which allows us to evaluate our model using stratified k-fold cross-validation more efficiently:

```
>>> from sklearn.cross_validation import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                          X=X_train,
...                          y=y_train,
...                          cv=10,
...                          n_jobs=1)
>>> print('CV accuracy scores: %s' % scores)
CV accuracy scores: [ 0.89130435  0.97826087  0.97826087
                      0.91304348  0.93478261  0.97777778
                      0.93333333  0.95555556  0.97777778
                        0.95555556]
>>> print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

An extremely useful feature of the `cross_val_score` approach is that we can distribute the evaluation of the different folds across multiple CPUs on our machine. If we set the `n_jobs` parameter to `1`, only one CPU will be used to evaluate the performances just like in our `StratifiedKFold` example previously. However, by setting `n_jobs=2` we could distribute the 10 rounds of cross-validation to two CPUs (if available on our machine), and by setting `n_jobs=-1`, we can use all available CPUs on our machine to do the computation in parallel.

Please note that a detailed discussion of how the variance of the generalization performance is estimated in cross-validation is beyond the scope of this book, but you can find a detailed discussion in this excellent article by M. Markatou et al (M. Markatou, H. Tian, S. Biswas, and G. M. Hripcsak. *Analysis of Variance of Cross-validation Estimators of the Generalization Error. Journal of Machine Learning Research*, 6:1127–1168, 2005).

You can also read about alternative cross-validation techniques, such as the .632 Bootstrap cross-validation method (B. Efron and R. Tibshirani. *Improvements on Cross-validation: The 632+ Bootstrap Method. Journal of the American Statistical Association*, 92(438):548–560, 1997).
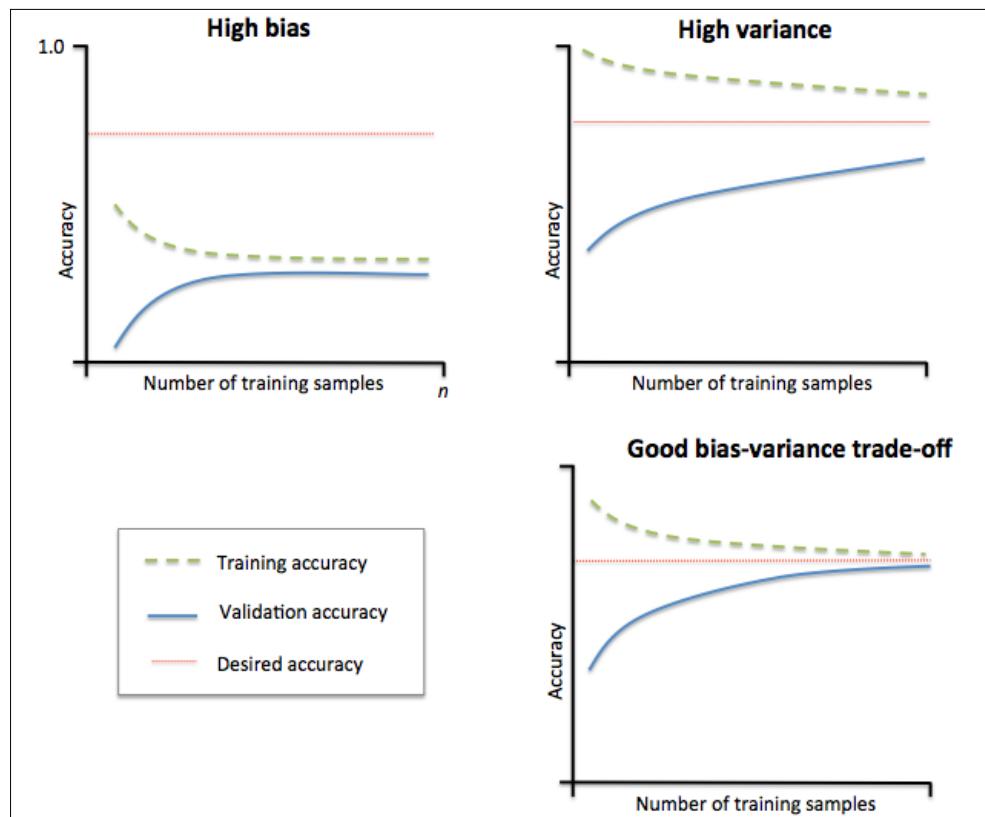
Debugging algorithms with learning and validation curves

In this section, we will take a look at two very simple yet powerful diagnostic tools that can help us to improve the performance of a learning algorithm: **learning curves** and **validation curves**. In the next subsections, we will discuss how we can use learning curves to diagnose if a learning algorithm has a problem with overfitting (high variance) or underfitting (high bias). Furthermore, we will take a look at validation curves that can help us address the common issues of a learning algorithm.

Diagnosing bias and variance problems with learning curves

If a model is too complex for a given training dataset—there are too many degrees of freedom or parameters in this model—the model tends to overfit the training data and does not generalize well to unseen data. Often, it can help to collect more training samples to reduce the degree of overfitting. However, in practice, it can often be very expensive or simply not feasible to collect more data. By plotting the model training and validation accuracies as functions of the training set size, we can easily detect whether the model suffers from high variance or high bias, and whether the collection of more data could help to address this problem. But before we discuss how to plot learning curves in sckit-learn, let's discuss those two common model issues by walking through the following illustration:
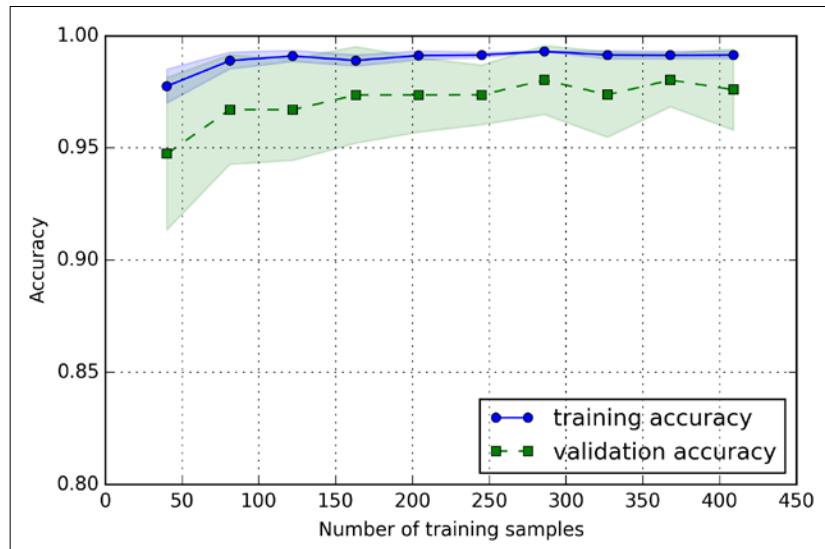
The graph in the upper-left shows a model with high bias. This model has both low training and cross-validation accuracy, which indicates that it underfits the training data. Common ways to address this issue are to increase the number of parameters of the model, for example, by collecting or constructing additional features, or by decreasing the degree of regularization, for example, in SVM or logistic regression classifiers. The graph in the upper-right shows a model that suffers from high variance, which is indicated by the large gap between the training and cross-validation accuracy. To address this problem of overfitting, we can collect more training data or reduce the complexity of the model, for example, by increasing the regularization parameter; for unregularized models, it can also help to decrease the number of features via feature selection (*Chapter 4*, *Building Good Training Sets – Data Preprocessing*) or feature extraction (*Chapter 5*, *Compressing Data via Dimensionality Reduction*). We shall note that collecting more training data decreases the chance of overfitting. However, it may not always help, for example, when the training data is extremely noisy or the model is already very close to optimal.

In the next subsection, we will see how to address those model issues using validation curves, but let's first see how we can use the learning curve function from scikit-learn to evaluate the model:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.learning_curve import learning_curve
>>> pipe_lr = Pipeline([
...             ('scl', StandardScaler()),
...             ('clf', LogisticRegression(
...                         penalty='l2', random_state=0))])
>>> train_sizes, train_scores, test_scores =\
...         learning_curve(estimator=pipe_lr,
...                         X=X_train,
...                         y=y_train,
...                         train_sizes=np.linspace(0.1, 1.0, 10),
...                         cv=10,
...                         n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(train_sizes, train_mean,
...         color='blue', marker='o',
...         markersize=5,
...         label='training accuracy')
>>> plt.fill_between(train_sizes,
...                 train_mean + train_std,
...                 train_mean - train_std,
```

```
...                        alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='validation accuracy')
>>> plt.fill_between(train_sizes,
...                     test_mean + test_std,
...                     test_mean - test_std,
...                     alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Number of training samples')
>>> plt.ylabel('Accuracy')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

After we have successfully executed the preceding code, we will obtain the following learning curve plot:

Via the `train_sizes` parameter in the `learning_curve` function, we can control the absolute or relative number of training samples that are used to generate the learning curves. Here, we set `train_sizes=np.linspace(0.1, 1.0, 10)` to use 10 evenly spaced relative intervals for the training set sizes. By default, the `learning_curve` function uses stratified k-fold cross-validation to calculate the cross-validation accuracy, and we set $k = 10$ via the `cv` parameter. Then, we simply calculate the average accuracies from the returned cross-validated training and test scores for the different sizes of the training set, which we plotted using matplotlib's `plot` function. Furthermore, we add the standard deviation of the average accuracies to the plot using the `fill_between` function to indicate the variance of the estimate.

As we can see in the preceding learning curve plot, our model performs quite well on the test dataset. However, it may be slightly overfitting the training data indicated by a relatively small, but visible, gap between the training and cross-validation accuracy curves.
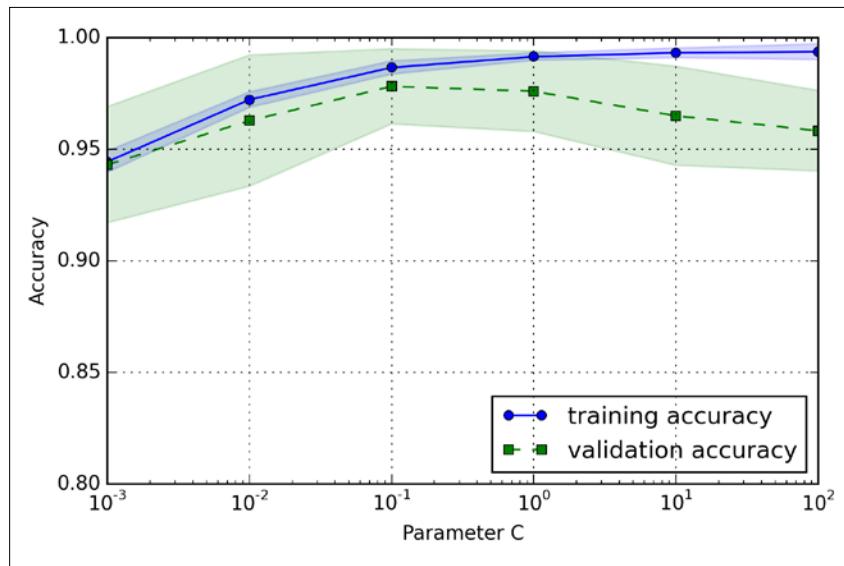
Addressing overfitting and underfitting with validation curves

Validation curves are a useful tool for improving the performance of a model by addressing issues such as overfitting or underfitting. Validation curves are related to learning curves, but instead of plotting the training and test accuracies as functions of the sample size, we vary the values of the model parameters, for example, the inverse regularization parameter `C` in logistic regression. Let's go ahead and see how we create validation curves via sckit-learn:

```
>>> from sklearn.learning_curve import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...                 estimator=pipe_lr,
...                  X=X_train,
...                  y=y_train,
...                  param_name='clf__C',
...                  param_range=param_range,
...                  cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
```

```
>>> plt.plot(param_range, train_mean,
...          color='blue', marker='o',
...          markersize=5,
...          label='training accuracy')
>>> plt.fill_between(param_range, train_mean + train_std,
...                  train_mean - train_std, alpha=0.15,
...                  color='blue')
>>> plt.plot(param_range, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='validation accuracy')
>>> plt.fill_between(param_range,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Parameter C')
>>> plt.ylabel('Accuracy')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

Using the preceding code, we obtained the validation curve plot for the parameter C:

Similar to the `learning_curve` function, the `validation_curve` function uses stratified k-fold cross-validation by default to estimate the performance of the model if we are using algorithms for classification. Inside the `validation_curve` function, we specified the parameter that we wanted to evaluate. In this case, it is `C`, the inverse regularization parameter of the `LogisticRegression` classifier, which we wrote as `'clf__C'` to access the `LogisticRegression` object inside the scikit-learn pipeline for a specified value range that we set via the `param_range` parameter. Similar to the learning curve example in the previous section, we plotted the average training and cross-validation accuracies and the corresponding standard deviations.

Although the differences in the accuracy for varying values of `C` are subtle, we can see that the model slightly underfits the data when we increase the regularization strength (small values of `C`). However, for large values of `C`, it means lowering the strength of regularization, so the model tends to slightly overfit the data. In this case, the sweet spot appears to be around `C=0.1`.

Fine-tuning machine learning models via grid search

In machine learning, we have two types of parameters: those that are learned from the training data, for example, the weights in logistic regression, and the parameters of a learning algorithm that are optimized separately. The latter are the tuning parameters, also called hyperparameters, of a model, for example, the **regularization** parameter in logistic regression or the **depth** parameter of a decision tree.

In the previous section, we used validation curves to improve the performance of a model by tuning one of its hyperparameters. In this section, we will take a look at a powerful hyperparameter optimization technique called **grid search** that can further help to improve the performance of a model by finding the *optimal* combination of hyperparameter values.

Tuning hyperparameters via grid search

The approach of grid search is quite simple, it's a brute-force exhaustive search paradigm where we specify a list of values for different hyperparameters, and the computer evaluates the model performance for each combination of those to obtain the optimal set:

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.svm import SVC
>>> pipe_svc = Pipeline([('scl', StandardScaler()),
...                      ('clf', SVC(random_state=1))])
>>> param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{'clf__C': param_range,
...                'clf__kernel': ['linear']},
...               {'clf__C': param_range,
...                'clf__gamma': param_range,
...                'clf__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=10,
...                   n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.978021978022
>>> print(gs.best_params_)
{'clf__C': 0.1, 'clf__kernel': 'linear'}
```

Using the preceding code, we initialized a `GridSearchCV` object from the `sklearn.grid_search` module to train and tune a **support vector machine** (**SVM**) pipeline. We set the `param_grid` parameter of `GridSearchCV` to a list of dictionaries to specify the parameters that we'd want to tune. For the linear SVM, we only evaluated the inverse regularization parameter `C`; for the RBF kernel SVM, we tuned both the `C` and `gamma` parameter. Note that the `gamma` parameter is specific to kernel SVMs. After we used the training data to perform the grid search, we obtained the score of the best-performing model via the `best_score_` attribute and looked at its parameters, that can be accessed via the `best_params_` attribute. In this particular case, the linear SVM model with `'clf__C'= 0.1'` yielded the best k-fold cross-validation accuracy: 97.8 percent.

Finally, we will use the independent test dataset to estimate the performance of the best selected model, which is available via the `best_estimator_` attribute of the `GridSearchCV` object:
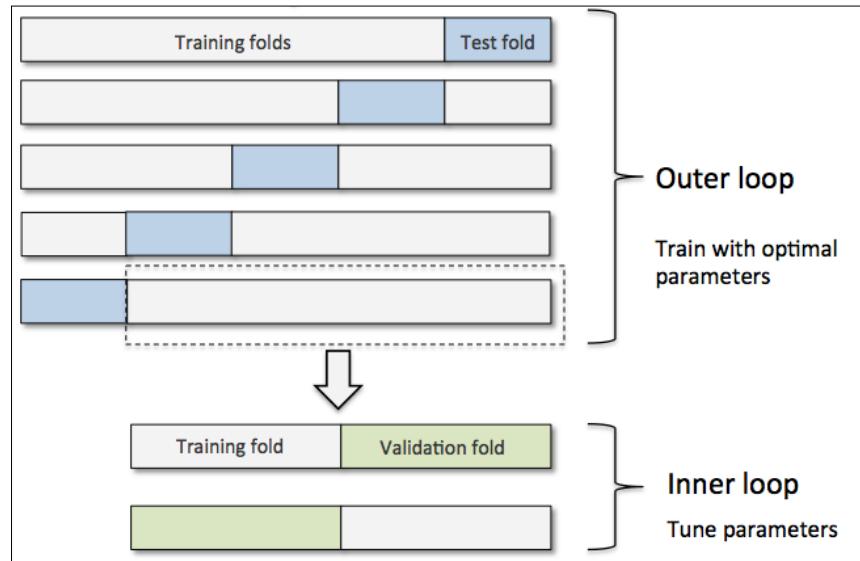
```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
>>> print('Test accuracy: %.3f' % clf.score(X_test, y_test))
Test accuracy: 0.965
```

> Although grid search is a powerful approach for finding the optimal set of parameters, the evaluation of all possible parameter combinations is also computationally very expensive. An alternative approach to sampling different parameter combinations using scikit-learn is randomized search. Using the `RandomizedSearchCV` class in scikit-learn, we can draw random parameter combinations from sampling distributions with a specified budget. More details and examples for its usage can be found at `http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization`.

Algorithm selection with nested cross-validation

Using k-fold cross-validation in combination with grid search is a useful approach for fine-tuning the performance of a machine learning model by varying its hyperparameters values as we saw in the previous subsection. If we want to select among different machine learning algorithms though, another recommended approach is nested cross-validation, and in a nice study on the bias in error estimation, Varma and Simon concluded that the true error of the estimate is almost unbiased relative to the test set when nested cross-validation is used (S. Varma and R. Simon. *Bias in Error Estimation When Using Cross-validation for Model Selection*. BMC bioinformatics, 7(1):91, 2006).

In nested cross-validation, we have an outer k-fold cross-validation loop to split the data into training and test folds, and an inner loop is used to select the model using k-fold cross-validation on the training fold. After model selection, the test fold is then used to evaluate the model performance. The following figure explains the concept of nested cross-validation with five outer and two inner folds, which can be useful for large data sets where computational performance is important; this particular type of nested cross-validation is also known as **5x2 cross-validation**:

In scikit-learn, we can perform nested cross-validation as follows:

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=2,
...                   n_jobs=-1)
>>> scores = cross_val_score(gs, X_train, y_train, scoring='accuracy',
cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...               np.mean(scores), np.std(scores)))
CV accuracy: 0.965 +/- 0.025
```

The returned average cross-validation accuracy gives us a good estimate of what to expect if we tune the hyperparameters of a model and then use it on unseen data. For example, we can use the nested cross-validation approach to compare an SVM model to a simple decision tree classifier; for simplicity, we will only tune its depth parameter:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...       estimator=DecisionTreeClassifier(random_state=0),
...       param_grid=[
...            {'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
...       scoring='accuracy',
...       cv=5)
>>> scores = cross_val_score(gs,
...                          X_train,
...                          y_train,
...                          scoring='accuracy',
...                          cv=2)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                      np.mean(scores), np.std(scores)))
CV accuracy: 0.921 +/- 0.029
```

As we can see here, the nested cross-validation performance of the SVM model (97.8 percent) is notably better than the performance of the decision tree (90.8 percent). Thus, we'd expect that it might be the better choice for classifying new data that comes from the same population as this particular dataset.

Looking at different performance evaluation metrics

In the previous sections and chapters, we evaluated our models using the model accuracy, which is a useful metric to quantify the performance of a model in general. However, there are several other performance metrics that can be used to measure a model's relevance, such as **precision**, **recall**, and the **F1-score**.

Reading a confusion matrix

Before we get into the details of different scoring metrics, let's print a so-called **confusion matrix**, a matrix that lays out the performance of a learning algorithm. The confusion matrix is simply a square matrix that reports the counts of the **true positive**, **true negative**, **false positive**, and **false negative** predictions of a classifier, as shown in the following figure:

Although these metrics can be easily computed manually by comparing the true and predicted class labels, scikit-learn provides a convenient `confusion_matrix` function that we can use as follows:
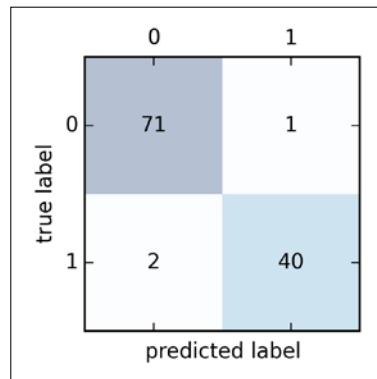
```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[71  1]
 [ 2 40]]
```

The array that was returned after executing the preceding code provides us with information about the different types of errors the classifier made on the test dataset that we can map onto the confusion matrix illustration in the previous figure using matplotlib's `matshow` function:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i,
...                 s=confmat[i, j],
...                 va='center', ha='center')
```

```
>>> plt.xlabel('predicted label')
>>> plt.ylabel('true label')
>>> plt.show()
```

Now, the confusion matrix plot as shown here should make the results a little bit easier to interpret:

Assuming that class 1 (malignant) is the positive class in this example, our model correctly classified 71 of the samples that belong to class 0 (true negatives) and 40 samples that belong to class 1 (true positives), respectively. However, our model also incorrectly misclassified 1 sample from class 0 as class 1 (false positive), and it predicted that 2 samples are benign although it is a malignant tumor (false negatives). In the next section, we will learn how we can use this information to calculate various different error metrics.

Optimizing the precision and recall of a classification model

Both the prediction **error** (**ERR**) and **accuracy** (**ACC**) provide general information about how many samples are misclassified. The error can be understood as the sum of all false predictions divided by the number of total predictions, and the accuracy is calculated as the sum of correct predictions divided by the total number of predictions, respectively:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

The prediction accuracy can then be calculated directly from the error:

$$ACC = \frac{TP+TN}{FP+FN+TP+TN} = 1 - ERR$$

The **true positive rate** (**TPR**) and **false positive rate** (**FPR**) are performance metrics that are especially useful for imbalanced class problems:

$$FPR = \frac{FP}{N} = \frac{FP}{FP+TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN+TP}$$

In tumor diagnosis, for example, we are more concerned about the detection of malignant tumors in order to help a patient with the appropriate treatment. However, it is also important to decrease the number of benign tumors that were incorrectly classified as malignant (false positives) to not unnecessarily concern a patient. In contrast to the FPR, the true positive rate provides useful information about the fraction of positive (or relevant) samples that were correctly identified out of the total pool of positives (**P**).

Precision (**PRE**) and **recall** (**REC**) are performance metrics that are related to those true positive and true negative rates, and in fact, recall is synonymous to the true positive rate:

$$PRE = \frac{TP}{TP+FP}$$

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN+TP}$$

In practice, often a combination of precision and recall is used, the so-called **F1-score**:

$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

These scoring metrics are all implemented in scikit-learn and can be imported from the `sklearn.metrics` module, as shown in the following snippet:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics  import recall_score, f1_score
>>> print('Precision: %.3f' % precision_score(
...               y_true=y_test, y_pred=y_pred))
Precision: 0.976
>>> print('Recall: %.3f' % recall_score(
...               y_true=y_test, y_pred=y_pred))
Recall: 0.952
>>> print('F1: %.3f' % f1_score(
...               y_true=y_test, y_pred=y_pred))
F1: 0.964
```

Furthermore, we can use a different scoring metric other than accuracy in `GridSearch` via the scoring parameter. A complete list of the different values that are accepted by the scoring parameter can be found at `http://scikit-learn.org/stable/modules/model_evaluation.html`.

Remember that the positive class in scikit-learn is the class that is labeled as class 1. If we want to specify a different *positive label*, we can construct our own scorer via the `make_scorer` function, which we can then directly provide as an argument to the scoring parameter in `GridSearchCV`:

```
>>> from sklearn.metrics import make_scorer, f1_score
>>> scorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                    param_grid=param_grid,
...                    scoring=scorer,
...                    cv=10)
```

Plotting a receiver operating characteristic

Receiver operator characteristic (**ROC**) graphs are useful tools for selecting models for classification based on their performance with respect to the false positive and true positive rates, which are computed by shifting the decision threshold of the classifier. The diagonal of an ROC graph can be interpreted as random guessing, and classification models that fall below the diagonal are considered as worse than random guessing. A perfect classifier would fall into the top-left corner of the graph with a true positive rate of 1 and a false positive rate of 0. Based on the ROC curve, we can then compute the so-called **area under the curve** (**AUC**) to characterize the performance of a classification model.

> Similar to ROC curves, we can compute **precision-recall curves** for the different probability thresholds of a classifier. A function for plotting those precision-recall curves is also implemented in scikit-learn and is documented at `http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html`.
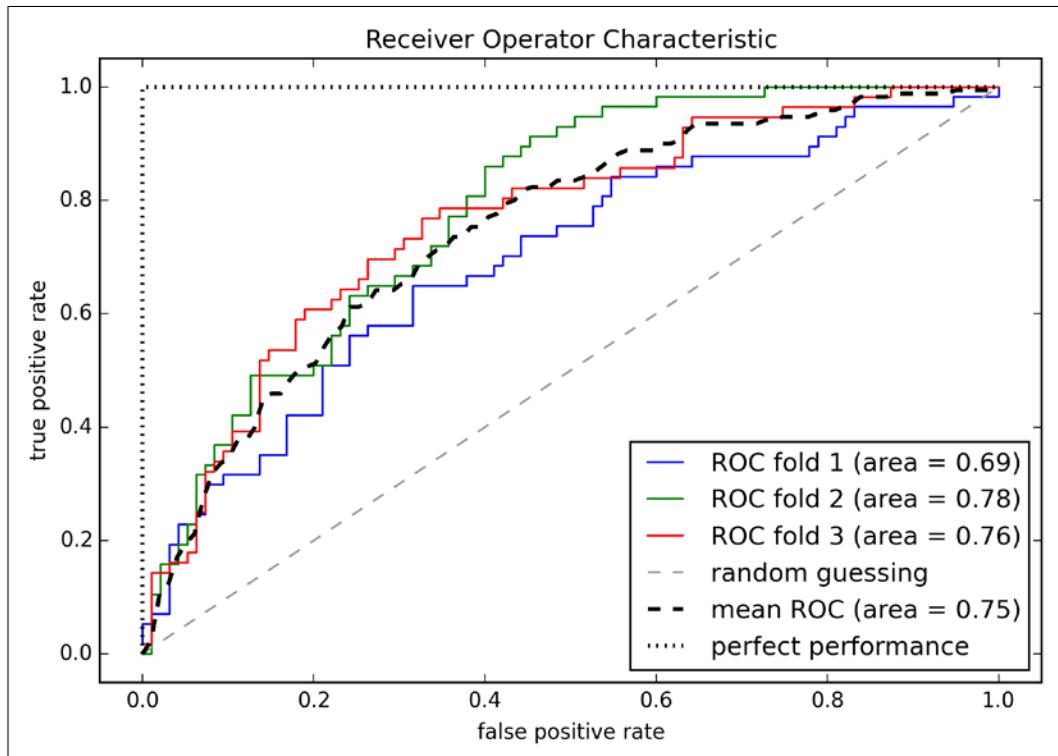
By executing the following code example, we will plot an ROC curve of a classifier that only uses two features from the Breast Cancer Wisconsin dataset to predict whether a tumor is benign or malignant. Although we are going to use the same logistic regression pipeline that we defined previously, we are making the classification task more challenging for the classifier so that the resulting ROC curve becomes visually more interesting. For similar reasons, we are also reducing the number of folds in the `StratifiedKFold` validator to three. The code is as follows:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...                     ('pca', PCA(n_components=2)),
...                     ('clf', LogisticRegression(penalty='l2',
...                                                random_state=0,
...                                                C=100.0))])
>>> X_train2 = X_train[:, [4, 14]]
>>> cv = StratifiedKFold(y_train,
...                      n_folds=3,
...                      random_state=1)
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []

>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(X_train2[train],
>>> y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
```

```
...                                                probas[:, 1],
...                                                pos_label=1)
...        mean_tpr += interp(mean_fpr, fpr, tpr)
...        mean_tpr[0] = 0.0
...        roc_auc = auc(fpr, tpr)
...        plt.plot(fpr,
...                  tpr,
...                  lw=1,
...                  label='ROC fold %d (area = %0.2f)'
...                        % (i+1, roc_auc))
>>> plt.plot([0, 1],
...          [0, 1],
...          linestyle='--',
...          color=(0.6, 0.6, 0.6),
...          label='random guessing')
>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...          label='mean ROC (area = %0.2f)' % mean_auc, lw=2)
>>> plt.plot([0, 0, 1],
...          [0, 1, 1],
...          lw=2,
...          linestyle=':',
...          color='black',
...          label='perfect performance')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('false positive rate')
>>> plt.ylabel('true positive rate')
>>> plt.title('Receiver Operator Characteristic')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

In the preceding code example, we used the already familiar `StratifiedKFold` class from scikit-learn and calculated the ROC performance of the `LogisticRegression` classifier in our `pipe_lr` pipeline using the `roc_curve` function from the `sklearn.metrics` module separately for each iteration. Furthermore, we interpolated the average ROC curve from the three folds via the `interp` function that we imported from SciPy and calculated the area under the curve via the `auc` function. The resulting ROC curve indicates that there is a certain degree of variance between the different folds, and the average ROC AUC (0.75) falls between a perfect score (1.0) and random guessing (0.5):

If we are just interested in the ROC AUC score, we could also directly import the `roc_auc_score` function from the `sklearn.metrics` submodule. The following code calculates the classifier's ROC AUC score on the independent test dataset after fitting it on the two-feature training set:

```
>>> pipe_lr = pipe_lr.fit(X_train2, y_train)
>>> y_pred2 = pipe_lr.predict(X_test[:, [4, 14]])
```

```
>>> from sklearn.metrics import roc_auc_score
>>> from sklearn.metrics import accuracy_score
>>> print('ROC AUC: %.3f' % roc_auc_score(
...          y_true=y_test, y_score=y_pred2))
ROC AUC: 0.662

>>> print('Accuracy: %.3f' % accuracy_score(
...          y_true=y_test, y_pred=y_pred2))
Accuracy: 0.711
```

Reporting the performance of a classifier as the ROC AUC can yield further insights in a classifier's performance with respect to imbalanced samples. However, while the accuracy score can be interpreted as a single cut-off point on a ROC curve, A. P. Bradley showed that the ROC AUC and accuracy metrics mostly agree with each other (A. P. Bradley. *The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms*. Pattern recognition, 30(7):1145–1159, 1997).

The scoring metrics for multiclass classification

The scoring metrics that we discussed in this section are specific to binary classification systems. However, scikit-learn also implements **macro** and **micro** averaging methods to extend those scoring metrics to multiclass problems via **One vs. All** (**OvA**) classification. The micro-average is calculated from the individual true positives, true negatives, false positives, and false negatives of the system. For example, the micro-average of the precision score in a k-class system can be calculated as follows:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

The macro-average is simply calculated as the average scores of the different systems:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Micro-averaging is useful if we want to weight each instance or prediction equally, whereas macro-averaging weights all classes equally to evaluate the overall performance of a classifier with regard to the most frequent class labels.

If we are using binary performance metrics to evaluate multiclass classification models in scikit-learn, a normalized or weighted variant of the macro-average is used by default. The weighted macro-average is calculated by weighting the score of each class label by the number of true instances when calculating the average. The weighted macro-average is useful if we are dealing with class imbalances, that is, different numbers of instances for each label.

While the weighted macro-average is the default for multiclass problems in scikit-learn, we can specify the averaging method via the `average` parameter inside the different scoring functions that we import from the `sklearn.metrics` module, for example, the `precision_score` or `make_scorer` functions:

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                          pos_label=1,
...                          greater_is_better=True,
...                          average='micro')
```

Summary

In the beginning of this chapter, we discussed how to chain different transformation techniques and classifiers in convenient model pipelines that helped us to train and evaluate machine learning models more efficiently. We then used those pipelines to perform k-fold cross-validation, one of the essential techniques for model selection and evaluation. Using k-fold cross-validation, we plotted learning and validation curves to diagnose the common problems of learning algorithms, such as overfitting and underfitting. Using grid search, we further fine-tuned our model. We concluded this chapter by looking at a confusion matrix and various different performance metrics that can be useful to further optimize a model's performance for a specific problem task. Now, we should be well-equipped with the essential techniques to build supervised machine learning models for classification successfully.

In the next chapter, we will take a look at ensemble methods, methods that allow us to combine multiple models and classification algorithms to boost the predictive performance of a machine learning system even further.

7
Combining Different Models for Ensemble Learning
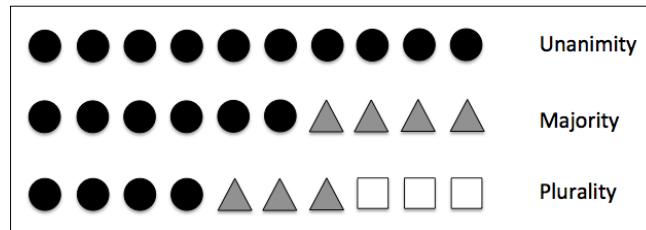
In the previous chapter, we focused on the best practices for tuning and evaluating different models for classification. In this chapter, we will build upon these techniques and explore different methods for constructing a set of classifiers that can often have a better predictive performance than any of its individual members. You will learn how to:

- Make predictions based on majority voting

- Reduce overfitting by drawing random combinations of the training set with repetition

- Build powerful models from *weak learners* that learn from their mistakes
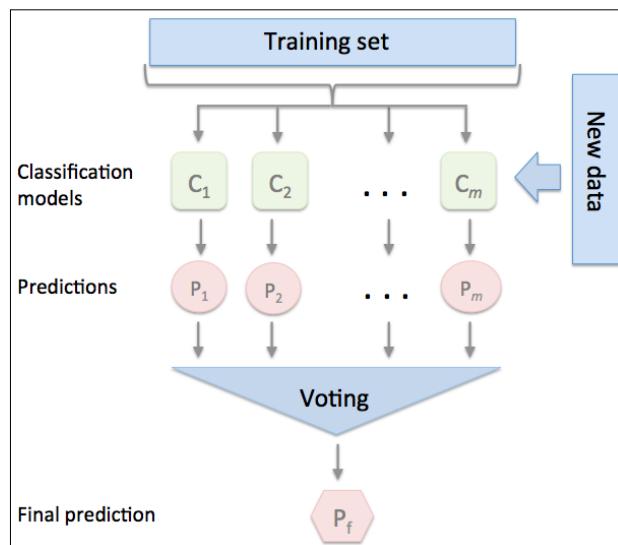
Learning with ensembles

The goal behind **ensemble methods** is to combine different classifiers into a meta-classifier that has a better generalization performance than each individual classifier alone. For example, assuming that we collected predictions from 10 experts, ensemble methods would allow us to strategically combine these predictions by the 10 experts to come up with a prediction that is more accurate and robust than the predictions by each individual expert. As we will see later in this chapter, there are several different approaches for creating an ensemble of classifiers. In this section, we will introduce a basic perception about how ensembles work and why they are typically recognized for yielding a good generalization performance.

In this chapter, we will focus on the most popular ensemble methods that use the **majority voting** principle. Majority voting simply means that we select the class label that has been predicted by the majority of classifiers, that is, received more than 50 percent of the votes. Strictly speaking, the term **majority vote** refers to binary class settings only. However, it is easy to generalize the majority voting principle to multi-class settings, which is called **plurality voting**. Here, we select the class label that received the most votes (mode). The following diagram illustrates the concept of majority and plurality voting for an ensemble of 10 classifiers where each unique symbol (triangle, square, and circle) represents a unique class label:

Using the training set, we start by training m different classifiers $(C_1,...,C_m)$. Depending on the technique, the ensemble can be built from different classification algorithms, for example, decision trees, support vector machines, logistic regression classifiers, and so on. Alternatively, we can also use the same base classification algorithm fitting different subsets of the training set. One prominent example of this approach would be the random forest algorithm, which combines different decision tree classifiers. The following diagram illustrates the concept of a general ensemble approach using majority voting:

To predict a class label via a simple majority or plurality voting, we combine the predicted class labels of each individual classifier C_j and select the class label $\hat{y}$ that received the most votes:

$$\hat{y} = mode\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

For example, in a binary classification task where $class1 = -1$ and $class2 = +1$, we can write the majority vote prediction as follows:

$$C(\boldsymbol{x}) = sign\left[\sum_j^m C_j(\boldsymbol{x})\right] = \begin{cases} 1 & if \sum_i C_j(\boldsymbol{x}) \geq 0 \\ -1 & otherwise \end{cases}$$

To illustrate why ensemble methods can work better than individual classifiers alone, let's apply the simple concepts of combinatorics. For the following example, we make the assumption that all n base classifiers for a binary classification task have an equal error rate ε. Furthermore, we assume that the classifiers are independent and the error rates are not correlated. Under those assumptions, we can simply express the error probability of an ensemble of base classifiers as a probability mass function of a binomial distribution:

$$P(y \geq k) = \sum_k^n \binom{n}{k} \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{ensemble}$$

Here, $\binom{n}{k}$ is the binomial coefficient *n choose k*. In other words, we compute the probability that the prediction of the ensemble is wrong. Now let's take a look at a more concrete example of 11 base classifiers ($n = 11$) with an error rate of 0.25 ($\varepsilon = 0.25$):

$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1-\varepsilon)^{11-k} = 0.034$$
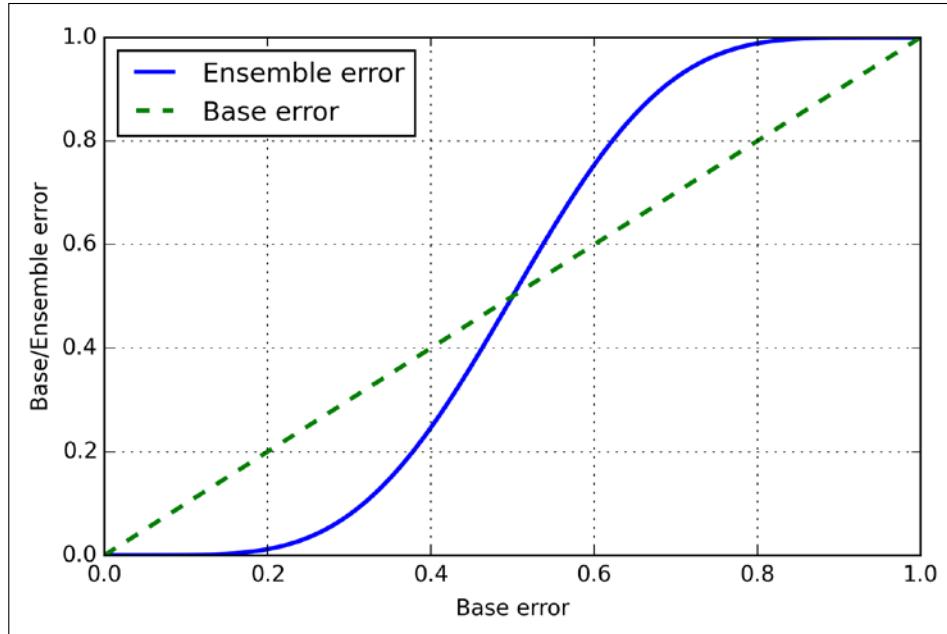
As we can see, the error rate of the ensemble (0.034) is much lower than the error rate of each individual classifier (0.25) if all the assumptions are met. Note that, in this simplified illustration, a 50-50 split by an even number of classifiers *n* is treated as an error, whereas this is only true half of the time. To compare such an idealistic ensemble classifier to a base classifier over a range of different base error rates, let's implement the probability mass function in Python:

```
>>> from scipy.misc import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = math.ceil(n_classifier / 2.0)
...     probs = [comb(n_classifier, k) *
...              error**k *
...              (1-error)**(n_classifier - k)
...              for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.034327507019042969
```

After we've implemented the `ensemble_error` function, we can compute the ensemble error rates for a range of different base errors from 0.0 to 1.0 to visualize the relationship between ensemble and base errors in a line graph:

```
>>> import numpy as np
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> import matplotlib.pyplot as plt
>>> plt.plot(error_range, ens_errors,
...          label='Ensemble error',
...          linewidth=2)
>>> plt.plot(error_range, error_range,
...          linestyle='--', label='Base error',
...          linewidth=2)
>>> plt.xlabel('Base error')
>>> plt.ylabel('Base/Ensemble error')
>>> plt.legend(loc='upper left')
>>> plt.grid()
>>> plt.show()
```

As we can see in the resulting plot, the error probability of an ensemble is always better than the error of an individual base classifier as long as the base classifiers perform better than random guessing ($\varepsilon < 0.5$). Note that the *y*-axis depicts the base error (dotted line) as well as the ensemble error (continuous line):

Implementing a simple majority vote classifier

After the short introduction to ensemble learning in the previous section, let's start with a warm-up exercise and implement a simple ensemble classifier for majority voting in Python. Although the following algorithm also generalizes to multi-class settings via plurality voting, we will use the term *majority voting* for simplicity as is also often done in literature.

The algorithm that we are going to implement will allow us to combine different classification algorithms associated with individual weights for confidence. Our goal is to build a stronger meta-classifier that balances out the individual classifiers' weaknesses on a particular dataset. In more precise mathematical terms, we can write the weighted majority vote as follows:

$$\hat{y} = \arg\max_{i} \sum_{j=1}^{m} w_j \chi_A \left(C_j \left(\boldsymbol{x} \right) = i \right)$$

Here, w_j is a weight associated with a base classifier, C_j, $\hat{y}$ is the predicted class label of the ensemble, χ_A (Greek *chi*) is the characteristic function $\left[C_j(x)=i \in A\right]$, and A is the set of unique class labels. For equal weights, we can simplify this equation and write it as follows:

$$\hat{y} = mode\left\{C_1(x), C_2(x), \ldots, C_m(x)\right\}$$

To better understand the concept of *weighting*, we will now take a look at a more concrete example. Let's assume that we have an ensemble of three base classifiers C_j ($j \in \{0,1\}$) and want to predict the class label of a given sample instance x. Two out of three base classifiers predict the class label 0, and one C_3 predicts that the sample belongs to class 1. If we weight the predictions of each base classifier equally, the majority vote will predict that the sample belongs to class 0:

$$C_1(x) \rightarrow 0,\ C_2(x) \rightarrow 0,\ C_3(x) \rightarrow 1$$

$$\hat{y} = mode\{0,0,1\} = 0$$

Now let's assign a weight of 0.6 to C_3 and weight C_1 and C_2 by a coefficient of 0.2, respectively.

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A\left(C_j(x)=i\right)$$

$$= \arg\max_i \left[0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1\right] = 1$$

More intuitively, since $3 \times 0.2 = 0.6$, we can say that the prediction made by C_3 has three times more weight than the predictions by C_1 or C_2, respectively. We can write this as follows:

$$\hat{y} = mode\{0,0,1,1,1\} = 1$$

To translate the concept of the weighted majority vote into Python code, we can use NumPy's convenient `argmax` and `bincount` functions:

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...           weights=[0.2, 0.2, 0.6]))
1
```

As discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, certain classifiers in scikit-learn can also return the probability of a predicted class label via the `predict_proba` method. Using the predicted class probabilities instead of the class labels for majority voting can be useful if the classifiers in our ensemble are well calibrated. The modified version of the majority vote for predicting class labels from probabilities can be written as follows:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$$

Here, p_{ij} is the predicted probability of the *jth* classifier for class label *i*.

To continue with our previous example, let's assume that we have a binary classification problem with class labels $i \in \{0,1\}$ and an ensemble of three classifiers C_j ($j \in \{1,2,3\}$). Let's assume that the classifier C_j returns the following class membership probabilities for a particular sample x:

$$C_1(x) \rightarrow [0.9, 0.1],\ C_2(x) \rightarrow [0.8, 0.2],\ C_3(x) \rightarrow [0.4, 0.6]$$

We can then calculate the individual class probabilities as follows:

$$p(i_0 \mid x) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1 \mid x) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.06 = 0.42$$

$$\hat{y} = \arg\max_i \left[p(i_0 \mid x), p(i_1 \mid x) \right] = 0$$

To implement the weighted majority vote based on class probabilities, we can again make use of NumPy using `numpy.average` and `np.argmax`:

```
>>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([ 0.58,  0.42])
>>> np.argmax(p)
0
```

Putting everything together, let's now implement a `MajorityVoteClassifier` in Python:

```python
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.externals import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator


class MajorityVoteClassifier(BaseEstimator,
                             ClassifierMixin):
    """ A majority vote ensemble classifier

    Parameters
    ----------
    classifiers : array-like, shape = [n_classifiers]
      Different classifiers for the ensemble

    vote : str, {'classlabel', 'probability'}
      Default: 'classlabel'
      If 'classlabel' the prediction is based on
      the argmax of class labels. Else if
      'probability', the argmax of the sum of
      probabilities is used to predict the class label
      (recommended for calibrated classifiers).

    weights : array-like, shape = [n_classifiers]
      Optional, default: None
      If a list of `int` or `float` values are
```

```
   provided, the classifiers are weighted by
   importance; Uses uniform weights if `weights=None`.

"""
def __init__(self, classifiers,
             vote='classlabel', weights=None):

    self.classifiers = classifiers
    self.named_classifiers = {key: value for
                                key, value in
                                _name_estimators(classifiers)}
    self.vote = vote
    self.weights = weights

def fit(self, X, y):
    """ Fit classifiers.

    Parameters
    ----------
    X : {array-like, sparse matrix},
        shape = [n_samples, n_features]
        Matrix of training samples.

    y : array-like, shape = [n_samples]
        Vector of target class labels.

    Returns
    -------
    self : object

    """
    # Use LabelEncoder to ensure class labels start
    # with 0, which is important for np.argmax
    # call in self.predict
    self.lablenc_ = LabelEncoder()
    self.lablenc_.fit(y)
    self.classes_ = self.lablenc_.classes_
    self.classifiers_ = []
    for clf in self.classifiers:
        fitted_clf = clone(clf).fit(X,
                          self.lablenc_.transform(y))
        self.classifiers_.append(fitted_clf)
    return self
```

I added a lot of comments to the code to better understand the individual parts. However, before we implement the remaining methods, let's take a quick break and discuss some of the code that may look confusing at first. We used the parent classes `BaseEstimator` and `ClassifierMixin` to get some base functionality *for free*, including the methods `get_params` and `set_params` to set and return the classifier's parameters as well as the `score` method to calculate the prediction accuracy, respectively. Also note that we imported `six` to make the `MajorityVoteClassifier` compatible with Python 2.7.

Next we will add the `predict` method to predict the class label via majority vote based on the class labels if we initialize a new `MajorityVoteClassifier` object with `vote='classlabel'`. Alternatively, we will be able to initialize the ensemble classifier with `vote='probability'` to predict the class label based on the class membership probabilities. Furthermore, we will also add a `predict_proba` method to return the average probabilities, which is useful to compute the **Receiver Operator Characteristic area under the curve (ROC AUC)**.

```python
def predict(self, X):
    """ Predict class labels for X.

    Parameters
    ----------
    X : {array-like, sparse matrix},
        Shape = [n_samples, n_features]
        Matrix of training samples.

    Returns
    ----------
    maj_vote : array-like, shape = [n_samples]
        Predicted class labels.

    """
    if self.vote == 'probability':
        maj_vote = np.argmax(self.predict_proba(X),
                             axis=1)
    else:  # 'classlabel' vote

        #  Collect results from clf.predict calls
        predictions = np.asarray([clf.predict(X)
                                  for clf in
                                  self.classifiers_]).T

        maj_vote = np.apply_along_axis(
                       lambda x:
                       np.argmax(np.bincount(x,
```

```
                                        weights=self.weights)),
                        axis=1,
                        arr=predictions)
    maj_vote = self.lablenc_.inverse_transform(maj_vote)
    return maj_vote

def predict_proba(self, X):
    """ Predict class probabilities for X.

    Parameters
    ----------
    X : {array-like, sparse matrix},
        shape = [n_samples, n_features]
        Training vectors, where n_samples is
        the number of samples and
        n_features is the number of features.

    Returns
    ----------
    avg_proba : array-like,
        shape = [n_samples, n_classes]
        Weighted average probability for
        each class per sample.

    """
    probas = np.asarray([clf.predict_proba(X)
                         for clf in self.classifiers_])
    avg_proba = np.average(probas,
                           axis=0, weights=self.weights)
    return avg_proba

def get_params(self, deep=True):
    """ Get classifier parameter names for GridSearch"""
    if not deep:
        return super(MajorityVoteClassifier,
                     self).get_params(deep=False)
    else:
        out = self.named_classifiers.copy()
        for name, step in\
                six.iteritems(self.named_classifiers):
            for key, value in six.iteritems(
                    step.get_params(deep=True)):
                out['%s__%s' % (name, key)] = value
        return out
```

Also, note that we defined our own modified version of the `get_params` methods to use the `_name_estimators` function in order to access the parameters of individual classifiers in the ensemble. This may look a little bit complicated at first, but it will make perfect sense when we use grid search for hyperparameter-tuning in later sections.

> Although our `MajorityVoteClassifier` implementation is very useful for demonstration purposes, I also implemented a more sophisticated version of the majority vote classifier in scikit-learn. It will become available as `sklearn.ensemble.VotingClassifier` in the next release version (v0.17).

Combining different algorithms for classification with majority vote

Now it is about time to put the `MajorityVoteClassifier` that we implemented in the previous section into action. But first, let's prepare a dataset that we can test it on. Since we are already familiar with techniques to load datasets from CSV files, we will take a shortcut and load the **Iris** dataset from scikit-learn's dataset module. Furthermore, we will only select two features, **sepal width** and **petal length**, to make the classification task more challenging. Although our `MajorityVoteClassifier` generalizes to multiclass problems, we will only classify flower samples from the two classes, **Iris-Versicolor** and **Iris-Virginica**, to compute the **ROC AUC**. The code is as follows:

```
>>> from sklearn import datasets
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

> Note that scikit-learn uses the `predict_proba` method (if applicable) to compute the ROC AUC score. In *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, we saw how the class probabilities are computed in logistic regression models. In decision trees, the probabilities are calculated from a frequency vector that is created for each node at training time. The vector collects the frequency values of each class label computed from the class label distribution at that node. Then the frequencies are normalized so that they sum up to 1. Similarly, the class labels of the k-nearest neighbors are aggregated to return the normalized class label frequencies in the k-nearest neighbors algorithm. Although the normalized probabilities returned by both the decision tree and k-nearest neighbors classifier may look similar to the probabilities obtained from a logistic regression model, we have to be aware that these are actually not derived from probability mass functions.

Next we split the Iris samples into 50 percent training and 50 percent test data:

```
>>> X_train, X_test, y_train, y_test =\
...         train_test_split(X, y,
...                          test_size=0.5,
...                          random_state=1)
```

Using the training dataset, we now will train three different classifiers—a logistic regression classifier, a decision tree classifier, and a k-nearest neighbors classifier—and look at their individual performances via a 10-fold cross-validation on the training dataset before we combine them into an ensemble classifier:

```
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           random_state=0)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])
```

```
>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                    ['clf', clf3]])
>>> clf_labels = ['Logistic Regression', 'Decision Tree', 'KNN']
>>> print('10-fold cross validation:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
>>>                              X=X_train,
>>>                              y=y_train,
>>>                              cv=10,
>>>                              scoring='roc_auc')
>>>     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...             % (scores.mean(), scores.std(), label))
```

The output that we receive, as shown in the following snippet, shows that the predictive performances of the individual classifiers are almost equal:

```
10-fold cross validation:

ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
```

You may be wondering why we trained the logistic regression and k-nearest neighbors classifier as part of a **pipeline**. The reason behind it is that, as discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, both the logistic regression and k-nearest neighbors algorithms (using the Euclidean distance metric) are not scale-invariant in contrast with decision trees. Although the Iris features are all measured on the same scale (cm), it is a good habit to work with standardized features.

Now let's move on to the more exciting part and combine the individual classifiers for majority rule voting in our `MajorityVoteClassifier`:

```
>>> mv_clf = MajorityVoteClassifier(
...               classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Majority Voting']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
...                              cv=10,
...                              scoring='roc_auc')
...     print("Accuracy: %0.2f (+/- %0.2f) [%s]"
...             % (scores.mean(), scores.std(), label))
```

```
ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
ROC AUC: 0.97 (+/- 0.10) [Majority Voting]
```
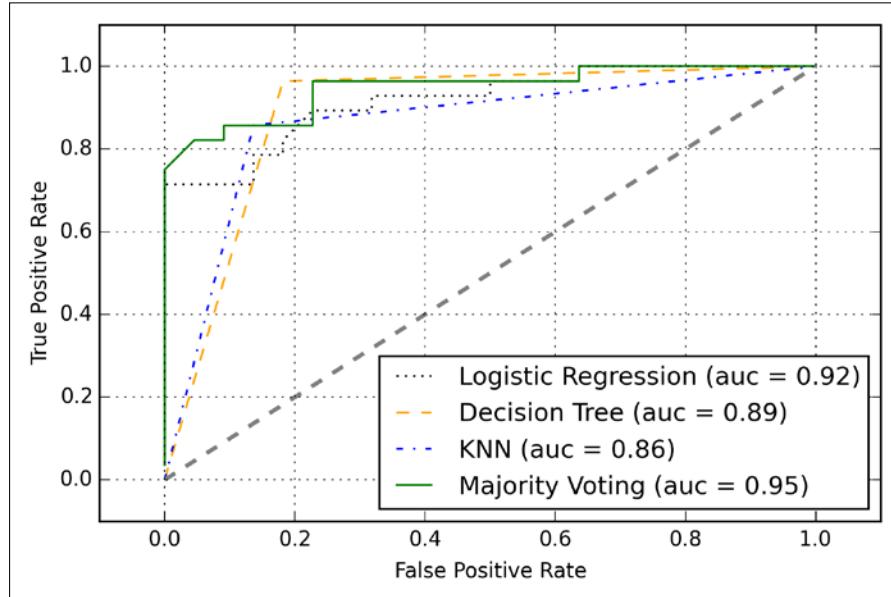
As we can see, the performance of the `MajorityVotingClassifier` has substantially improved over the individual classifiers in the 10-fold cross-validation evaluation.

Evaluating and tuning the ensemble classifier

In this section, we are going to compute the ROC curves from the test set to check if the `MajorityVoteClassifier` generalizes well to unseen data. We should remember that the test set is not to be used for model selection; its only purpose is to report an unbiased estimate of the generalization performance of a classifier system. The code is as follows:

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyles = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...         in zip(all_clf, clf_labels, colors, linestyles):
...     # assuming the label of the positive class is 1
...     y_pred = clf.fit(X_train,
...                      y_train).predict_proba(X_test)[:, 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                      y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...              color=clr,
...              linestyle=ls,
...              label='%s (auc = %0.2f)' % (label, roc_auc))
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...          linestyle='--',
...          color='gray',
...          linewidth=2)
>>> plt.xlim([-0.1, 1.1])
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid()
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
>>> plt.show()
```

As we can see in the resulting ROC, the ensemble classifier also performs well on the test set (*ROC AUC = 0.95*), whereas the k-nearest neighbors classifier seems to be overfitting the training data (training *ROC AUC = 0.93*, test *ROC AUC = 0.86*):
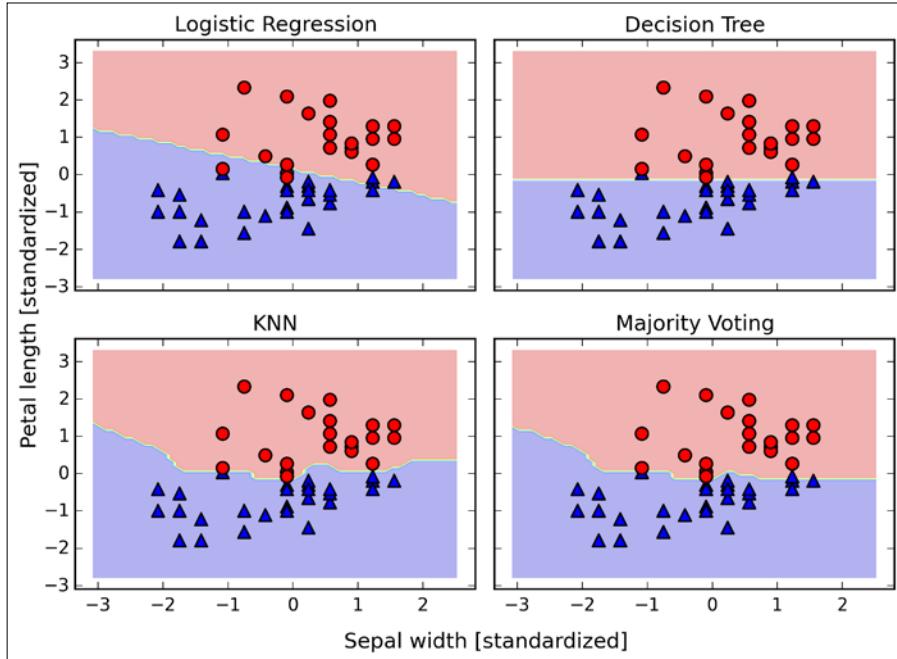
Since we only selected two features for the classification examples, it would be interesting to see what the decision region of the ensemble classifier actually looks like. Although it is not necessary to standardize the training features prior to model fitting because our logistic regression and k-nearest neighbors pipelines will automatically take care of this, we will standardize the training set so that the decision regions of the decision tree will be on the same scale for visual purposes. The code is as follows:

```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>> y_max = X_train_std[:, 1].max() + 1
```

```
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                         all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                   X_train_std[y_train==0, 1],
...                                   c='blue',
...                                   marker='^',
...                                   s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                   X_train_std[y_train==1, 1],
...                                   c='red',
...                                   marker='o',
...                                   s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -4.5,
...          s='Sepal width [standardized]',
...          ha='center', va='center', fontsize=12)
>>> plt.text(-10.5, 4.5,
...          s='Petal length [standardized]',
...          ha='center', va='center',
...          fontsize=12, rotation=90)
>>> plt.show()
```

Interestingly but also as expected, the decision regions of the ensemble classifier seem to be a hybrid of the decision regions from the individual classifiers. At first glance, the majority vote decision boundary looks a lot like the decision boundary of the k-nearest neighbor classifier. However, we can see that it is orthogonal to the y axis for *sepal width* ≥ 1, just like the decision tree stump:

Before you learn how to tune the individual classifier parameters for ensemble classification, let's call the `get_params` method to get a basic idea of how we can access the individual parameters inside a `GridSearch` object:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier': DecisionTreeClassifier(class_weight=None,
criterion='entropy', max_depth=1,
            max_features=None, max_leaf_nodes=None, min_samples_
leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            random_state=0, splitter='best'),
 'decisiontreeclassifier__class_weight': None,
 'decisiontreeclassifier__criterion': 'entropy',
 [...]
 'decisiontreeclassifier__random_state': 0,
 'decisiontreeclassifier__splitter': 'best',
```

```
 'pipeline-1': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', LogisticRegression(C=0.001, class_
weight=None, dual=False, fit_intercept=True,
           intercept_scaling=1, max_iter=100, multi_class='ovr',
           penalty='l2', random_state=0, solver='liblinear',
tol=0.0001,
           verbose=0))]),
 'pipeline-1__clf': LogisticRegression(C=0.001, class_weight=None,
dual=False, fit_intercept=True,
           intercept_scaling=1, max_iter=100, multi_class='ovr',
           penalty='l2', random_state=0, solver='liblinear',
tol=0.0001,
           verbose=0),
 'pipeline-1__clf__C': 0.001,
 'pipeline-1__clf__class_weight': None,
 'pipeline-1__clf__dual': False,
 [...]
 'pipeline-1__sc__with_std': True,
 'pipeline-2': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', KNeighborsClassifier(algorithm='au
to', leaf_size=30, metric='minkowski',
           metric_params=None, n_neighbors=1, p=2,
weights='uniform'))]),
 'pipeline-2__clf': KNeighborsClassifier(algorithm='auto', leaf_
size=30, metric='minkowski',
           metric_params=None, n_neighbors=1, p=2,
weights='uniform'),
 'pipeline-2__clf__algorithm': 'auto',
 [...]
 'pipeline-2__sc__with_std': True}
```

Based on the values returned by the `get_params` method, we now know how to access the individual classifier's attributes. Let's now tune the inverse regularization parameter `C` of the logistic regression classifier and the decision tree depth via a grid search for demonstration purposes. The code is as follows:

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...              'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                      param_grid=params,
...                      cv=10,
...                      scoring='roc_auc')
>>> grid.fit(X_train, y_train)
```

After the grid search has completed, we can print the different hyperparameter value combinations and the average ROC AUC scores computed via 10-fold cross-validation. The code is as follows:

```
>>> for params, mean_score, scores in grid.grid_scores_:
...     print("%0.3f+/-%0.2f %r"
...            % (mean_score, scores.std() / 2, params))
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__
max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_
depth': 1}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__
max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__
max_depth': 2}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_
depth': 2}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__
max_depth': 2}

>>> print('Best parameters: %s' % grid.best_params_)
Best parameters: {'pipeline-1__clf__C': 100.0,
'decisiontreeclassifier__max_depth': 1}

>>> print('Accuracy: %.2f' % grid.best_score_)
Accuracy: 1.00
```
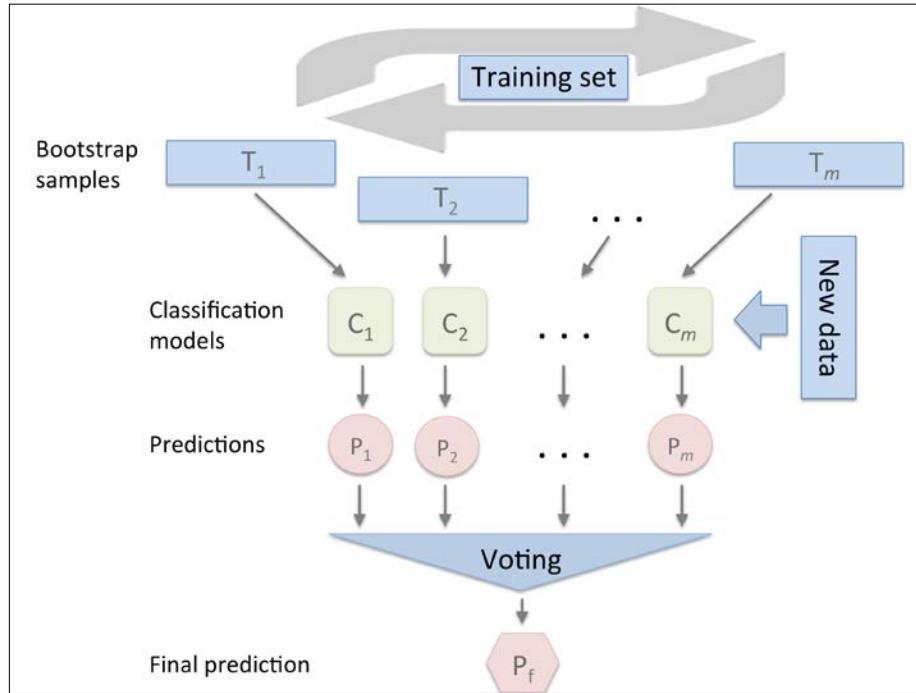
As we can see, we get the best cross-validation results when we choose a lower regularization strength (`C = 100.0`) whereas the tree depth does not seem to affect the performance at all, suggesting that a decision stump is sufficient to separate the data. To remind ourselves that it is a bad practice to use the test dataset more than once for model evaluation, we are not going to estimate the generalization performance of the tuned hyperparameters in this section. We will move on swiftly to an alternative approach for ensemble learning: **bagging**.

> The majority vote approach we implemented in this section is sometimes also referred to as **stacking**. However, the stacking algorithm is more typically used in combination with a logistic regression model that predicts the final class label using the predictions of the individual classifiers in the ensemble as input, which has been described in more detail by David H. Wolpert in D. H. Wolpert. *Stacked generalization*. Neural networks, 5(2):241–259, 1992.

Bagging – building an ensemble of classifiers from bootstrap samples

Bagging is an ensemble learning technique that is closely related to the
`MajorityVoteClassifier` that we implemented in the previous section,
as illustrated in the following diagram:

However, instead of using the same training set to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training set, which is why bagging is also known as **bootstrap aggregating**. To provide a more concrete example of how bootstrapping works, let's consider the example shown in the following figure. Here, we have seven different training instances (denoted as indices 1-7) that are sampled randomly with replacement in each round of bagging. Each bootstrap sample is then used to fit a classifier C_j, which is most typically an unpruned decision tree:

Sample indices	Bagging round 1	Bagging round 2	...
1	2	7	...
2	2	3	...
3	1	2	...
4	3	1	...
5	7	1	...
6	2	7	...
7	4	7	...
	C_1	C_2	C_m

Bagging is also related to the random forest classifier that we introduced in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*. In fact, random forests are a special case of bagging where we also use random feature subsets to fit the individual decision trees. Bagging was first proposed by Leo Breiman in a technical report in 1994; he also showed that bagging can improve the accuracy of unstable models and decrease the degree of overfitting. I highly recommend you read about his research in L. Breiman. *Bagging Predictors*. Machine Learning, 24(2):123–140, 1996, which is freely available online, to learn more about bagging.

To see bagging in action, let's create a more complex classification problem using the **Wine** dataset that we introduced in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*. Here, we will only consider the Wine classes 2 and 3, and we select two features: **Alcohol** and **Hue**.

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash',
...                    'Magnesium', 'Total phenols',
...                    'Flavanoids', 'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> df_wine = df_wine[df_wine['Class label'] != 1]
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol', 'Hue']].values
```

Next we encode the class labels into binary format and split the dataset into 60 percent training and 40 percent test set, respectively:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.cross_validation import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test =\
...          train_test_split(X, y,
...                           test_size=0.40,
...                           random_state=1)
```

A `BaggingClassifier` algorithm is already implemented in scikit-learn, which we can import from the `ensemble` submodule. Here, we will use an unpruned decision tree as the base classifier and create an ensemble of 500 decision trees fitted on different bootstrap samples of the training dataset:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=None,
...                               random_state=1)
>>> bag = BaggingClassifier(base_estimator=tree,
```

```
...                                n_estimators=500,
...                                max_samples=1.0,
...                                max_features=1.0,
...                                bootstrap=True,
...                                bootstrap_features=False,
...                                n_jobs=1,
...                                random_state=1)
```

Next we will calculate the accuracy score of the prediction on the training and test dataset to compare the performance of the bagging classifier to the performance of a single unpruned decision tree:

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...         % (tree_train, tree_test))
Decision tree train/test accuracies 1.000/0.833
```

Based on the accuracy values that we printed by executing the preceding code snippet, the unpruned decision tree predicts all class labels of the training samples correctly; however, the substantially lower test accuracy indicates high variance (overfitting) of the model:

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Bagging train/test accuracies %.3f/%.3f'
...         % (bag_train, bag_test))
Bagging train/test accuracies 1.000/0.896
```

Although the training accuracies of the decision tree and bagging classifier are similar on the training set (both 1.0), we can see that the bagging classifier has a slightly better generalization performance as estimated on the test set. Next let's compare the decision regions between the decision tree and bagging classifier:
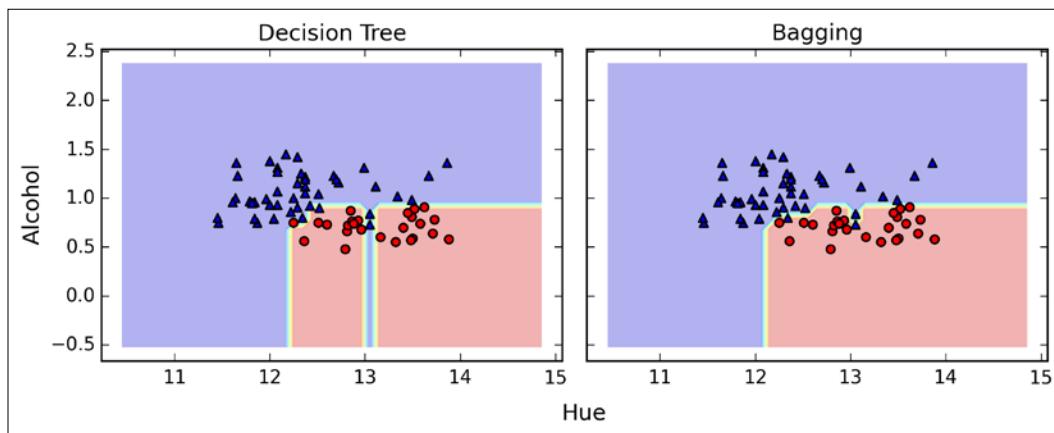
```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
```

```
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, bag],
...                         ['Decision Tree', 'Bagging']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                         X_train[y_train==0, 1],
...                         c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                         X_train[y_train==1, 1],
...                         c='red', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel(Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s=Hue',
...          ha='center', va='center', fontsize=12)
>>> plt.show()
```

As we can see in the resulting plot, the piece-wise linear decision boundary of the three-node deep decision tree looks smoother in the bagging ensemble:

We only looked at a very simple bagging example in this section. In practice, more complex classification tasks and datasets' high dimensionality can easily lead to overfitting in single decision trees and this is where the bagging algorithm can really play out its strengths. Finally, we shall note that the bagging algorithm can be an effective approach to reduce the variance of a model. However, bagging is ineffective in reducing model bias, which is why we want to choose an ensemble of classifiers with low bias, for example, unpruned decision trees.

Leveraging weak learners via adaptive boosting

In this section about ensemble methods, we will discuss **boosting** with a special focus on its most common implementation, **AdaBoost** (short for **Adaptive Boosting**).

> The original idea behind AdaBoost was formulated by Robert Schapire in 1990 (R. E. Schapire. *The Strength of Weak Learnability*. Machine learning, 5(2):197–227, 1990). After Robert Schapire and Yoav Freund presented the AdaBoost algorithm in the Proceedings of the Thirteenth International Conference (ICML 1996), AdaBoost became one of the most widely used ensemble methods in the years that followed (Y. Freund, R. E. Schapire, et al. *Experiments with a New Boosting Algorithm*. In ICML, volume 96, pages 148–156, 1996). In 2003, Freund and Schapire received the *Goedel Prize* for their groundbreaking work, which is a prestigious prize for the most outstanding publications in the computer science field.
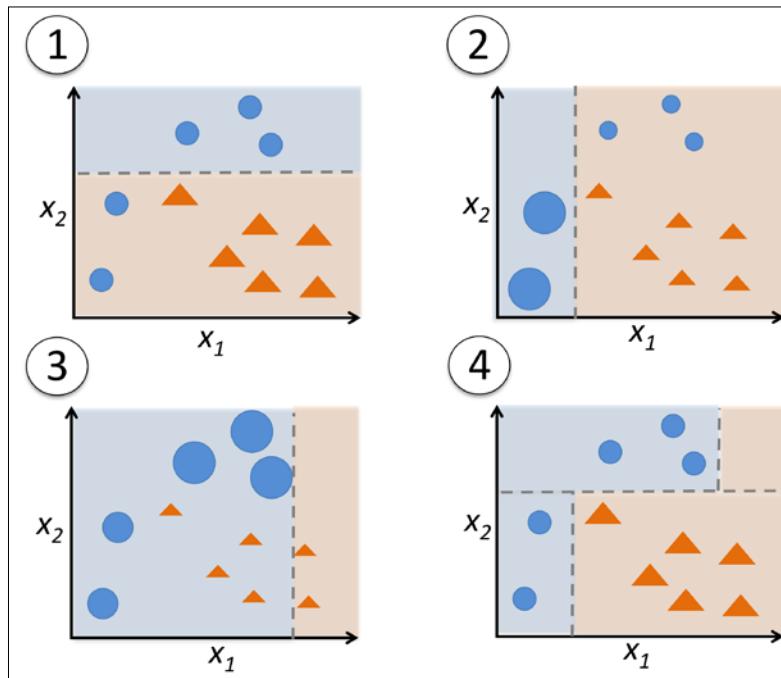
In boosting, the ensemble consists of very simple base classifiers, also often referred to as **weak learners**, that have only a slight performance advantage over random guessing. A typical example of a weak learner would be a decision tree stump. The key concept behind boosting is to focus on training samples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training samples to improve the performance of the ensemble. In contrast to bagging, the initial formulation of boosting, the algorithm uses random subsets of training samples drawn from the training dataset without replacement. The original boosting procedure is summarized in four key steps as follows:

1. Draw a random subset of training samples d_1 without replacement from the training set D to train a weak learner C_1.

2. Draw second random training subset d_2 without replacement from the training set and add 50 percent of the samples that were previously misclassified to train a weak learner C_2.

3. Find the training samples d_3 in the training set D on which C_1 and C_2 disagree to train a third weak learner C_3.

4. Combine the weak learners C_1, C_2, and C_3 via majority voting.

As discussed by Leo Breiman (L. Breiman. *Bias, Variance, and Arcing Classifiers*. 1996), boosting can lead to a decrease in bias as well as variance compared to bagging models. In practice, however, boosting algorithms such as AdaBoost are also known for their high variance, that is, the tendency to overfit the training data (G. Raetsch, T. Onoda, and K. R. Mueller. *An Improvement of Adaboost to Avoid Overfitting*. In Proc. of the Int. Conf. on Neural Information Processing. Citeseer, 1998).

In contrast to the original boosting procedure as described here, AdaBoost uses the complete training set to train the weak learners where the training samples are reweighted in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble. Before we dive deeper into the specific details of the AdaBoost algorithm, let's take a look at the following figure to get a better grasp of the basic concept behind AdaBoost:

To walk through the AdaBoost illustration step by step, we start with subfigure **1**, which represents a training set for binary classification where all training samples are assigned equal weights. Based on this training set, we train a decision stump (shown as a dashed line) that tries to classify the samples of the two classes (triangles and circles) as well as possible by minimizing the cost function (or the impurity score in the special case of decision tree ensembles). For the next round (subfigure **2**), we assign a larger weight to the two previously misclassified samples (circles). Furthermore, we lower the weight of the correctly classified samples. The next decision stump will now be more focused on the training samples that have the largest weights, that is, the training samples that are supposedly hard to classify. The weak learner shown in subfigure **2** misclassifies three different samples from the circle-class, which are then assigned a larger weight as shown in subfigure **3**. Assuming that our AdaBoost ensemble only consists of three rounds of boosting, we would then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote, as shown in subfigure **4**.

Now that have a better understanding behind the basic concept of AdaBoost, let's take a more detailed look at the algorithm using pseudo code. For clarity, we will denote element-wise multiplication by the cross symbol ($\times$) and the dot product between two vectors by a dot symbol ($\cdot$), respectively. The steps are as follows:

1. Set weight vector $\boldsymbol{w}$ to uniform weights where $\sum_i w_i = 1$
2. For j in m boosting rounds, do the following:
3. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
4. Predict class labels: $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$.
5. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{y} == \boldsymbol{y})$.
6. Compute coefficient: $\alpha_j = 0.5\log\dfrac{1-\varepsilon}{\varepsilon}$.
7. Update weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp\left(-\alpha_j \times \hat{y} \times \boldsymbol{y}\right)$.
8. Normalize weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
9. Compute final prediction: $\hat{y} = \left(\sum_{j=1}^{m}\left(\alpha_j \times \text{predict}(C_j, \boldsymbol{X})\right) > 0\right)$.

Note that the expression $(\hat{y} == \boldsymbol{y})$ in step 5 refers to a vector of 1s and 0s, where a 1 is assigned if the prediction is incorrect and 0 is assigned otherwise.

Although the AdaBoost algorithm seems to be pretty straightforward, let's walk through a more concrete example using a training set consisting of 10 training samples as illustrated in the following table:

Sample indices	x	y	Weights	$\hat{y}(x \le 3.0)$?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

The first column of the table depicts the sample indices of the training samples 1 to 10. In the second column, we see the feature values of the individual samples assuming this is a one-dimensional dataset. The third column shows the true class label y_i for each training sample x_i, where $y_i \in \{1, -1\}$. The initial weights are shown in the fourth column; we initialize the weights to uniform and normalize them to sum to one. In the case of the 10 sample training set, we therefore assign the 0.1 to each weight w_i in the weight vector w. The predicted class labels $\hat{y}$ are shown in the fifth column, assuming that our splitting criterion is $x \le 3.0$. The last column of the table then shows the updated weights based on the update rules that we defined in the pseudocode.

Since the computation of the weight updates may look a little bit complicated at first, we will now follow the calculation step by step. We start by computing the weighted error rate ε as described in step 5:

$$\varepsilon = 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 1 + 0.1 \times 1$$

$$+ 0.1 \times 1 + 0.1 \times 0 = \frac{3}{10} = 0.3$$

Next we compute the coefficient α_j (shown in step 6), which is later used in step 7 to update the weights as well as for the weights in majority vote prediction (step 10):

$$\alpha_j = 0.5 \log\left(\frac{1 - \varepsilon}{\varepsilon}\right) \approx 0.424$$

After we have computed the coefficient α_j we can now update the weight vector using the following equation:

$$w := w \times \exp\left(-\alpha_j \times \hat{y} \times y\right)$$

Here, $\hat{y} \times y$ is an element-wise multiplication between the vectors of the predicted and true class labels, respectively. Thus, if a prediction $\hat{y}_i$ is correct, $\hat{y}_i \times y_i$ will have a positive sign so that we decrease the *ith* weight since α_j is a positive number as well:

$$0.1 \times \exp\left(-0.424 \times 1 \times 1\right) \approx 0.065$$

Similarly, we will increase the *ith* weight if $\hat{y}_i$ predicted the label incorrectly like this:

$$0.1 \times \exp\left(-0.424 \times 1 \times \left(-1\right)\right) \approx 0.153$$

Or like this:

$$0.1 \times \exp\left(-0.424 \times \left(-1\right) \times \left(1\right)\right) \approx 0.153$$

After we update each weight in the weight vector, we normalize the weights so that they sum up to 1 (step 8):

$$w := \frac{w}{\sum_i w_i}$$

Here, $\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$.

Thus, each weight that corresponds to a correctly classified sample will be reduced from the initial value of 0.1 to $0.065 / 0.914 \approx 0.071$ for the next round of boosting. Similarly, the weights of each incorrectly classified sample will increase from 0.1 to $0.153 / 0.914 \approx 0.167$.

This was AdaBoost in a nutshell. Skipping to the more practical part, let's now train an AdaBoost ensemble classifier via scikit-learn. We will use the same Wine subset that we used in the previous section to train the bagging meta-classifier. Via the `base_estimator` attribute, we will train the `AdaBoostClassifier` on 500 decision tree stumps:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=None,
...                               random_state=0)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=0)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.845/0.854
```

As we can see, the decision tree stump tends to underfit the training data in contrast with the unpruned decision tree that we saw in the previous section:

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('AdaBoost train/test accuracies %.3f/%.3f'
...       % (ada_train, ada_test))
AdaBoost train/test accuracies 1.000/0.875
```
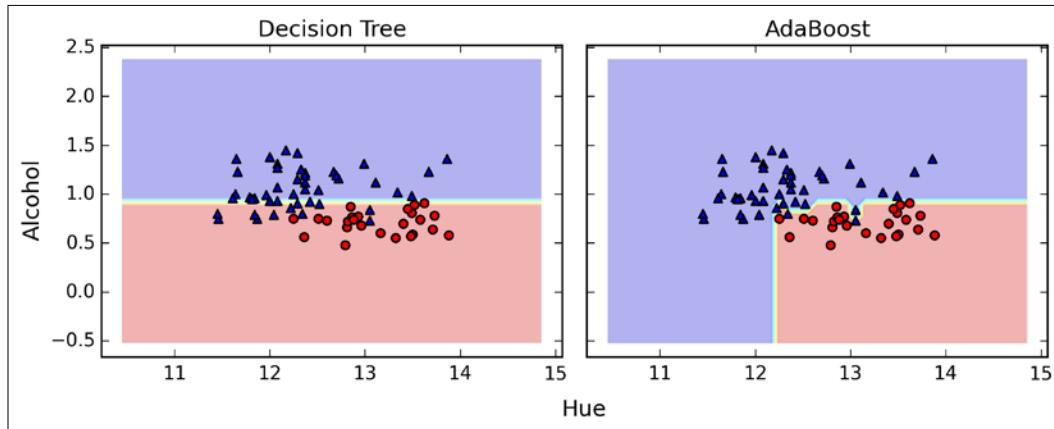
As we can see, the AdaBoost model predicts all class labels of the training set correctly and also shows a slightly improved test set performance compared to the decision tree stump. However, we also see that we introduced additional variance by our attempt to reduce the model bias.

Although we used another simple example for demonstration purposes, we can see that the performance of the AdaBoost classifier is slightly improved compared to the decision stump and achieved very similar accuracy scores to the bagging classifier that we trained in the previous section. However, we should note that it is considered bad practice to select a model based on the repeated usage of the test set. The estimate of the generalization performance may be too optimistic, which we discussed in more detail in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Finally, let's check what the decision regions look like:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
...                         ['Decision Tree', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red',
...                        marker='o')
... axarr[idx].set_title(tt)
... axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s=Hue',
...          ha='center',
...          va='center',
...          fontsize=12)
>>> plt.show()
```

By looking at the decision regions, we can see that the decision boundary of the AdaBoost model is substantially more complex than the decision boundary of the decision stump. In addition, we note that the AdaBoost model separates the feature space very similarly to the bagging classifier that we trained in the previous section.

As concluding remarks about ensemble techniques, it is worth noting that ensemble learning increases the computational complexity compared to individual classifiers. In practice, we need to think carefully whether we want to pay the price of increased computational costs for an often relatively modest improvement of predictive performance.

An often-cited example of this trade-off is the famous *$1 Million Netflix Prize*, which was won using ensemble techniques. The details about the algorithm were published in A. Toescher, M. Jahrer, and R. M. Bell. *The Bigchaos Solution to the Netflix Grand Prize*. Netflix prize documentation, 2009 (which is available at `http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf`). Although the winning team received the $1 million prize money, Netflix never implemented their model due to its complexity, which made it unfeasible for a real-world application. To quote their exact words (`http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`):

> *"[…] additional accuracy gains that we measured did not seem to justify the engineering effort needed to bring them into a production environment."*

Summary

In this chapter, we looked at some of the most popular and widely used techniques for ensemble learning. Ensemble methods combine different classification models to cancel out their individual weaknesses, which often results in stable and well-performing models that are very attractive for industrial applications as well as machine learning competitions.

In the beginning of this chapter, we implemented a `MajorityVoteClassifier` in Python that allows us to combine different algorithms for classification. We then looked at bagging, a useful technique to reduce the variance of a model by drawing random bootstrap samples from the training set and combining the individually trained classifiers via majority vote. Then we discussed AdaBoost, which is an algorithm that is based on weak learners that subsequently learn from mistakes.

Throughout the previous chapters, we discussed different learning algorithms, tuning, and evaluation techniques. In the following chapter, we will look at a particular application of machine learning, sentiment analysis, which has certainly become an interesting topic in the era of the Internet and social media.

8

Applying Machine Learning to Sentiment Analysis

In this Internet and social media time and age, people's opinions, reviews, and recommendations have become a valuable resource for political science and businesses. Thanks to modern technologies, we are now able to collect and analyze such data most efficiently. In this chapter, we will delve into a subfield of **natural language processing** (**NLP**) called **sentiment analysis** and learn how to use machine learning algorithms to classify documents based on their polarity: the attitude of the writer. The topics that we will cover in the following sections include:

- Cleaning and preparing text data
- Building feature vectors from text documents
- Training a machine learning model to classify positive and negative movie reviews
- Working with large text datasets using *out-of-core* learning

Obtaining the IMDb movie review dataset

Sentiment analysis, sometimes also called **opinion mining**, is a popular sub-discipline of the broader field of NLP; it analyzes the **polarity** of documents. A popular task in sentiment analysis is the classification of documents based on the expressed opinions or emotions of the authors with regard to a particular topic.

In this chapter, we will be working with a large dataset of movie reviews from the **Internet Movie Database** (**IMDb**) that has been collected by Maas et al. (A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. *Learning Word Vectors for Sentiment Analysis*. In the proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics). The movie review dataset consists of 50,000 polar movie reviews that are labeled as either *positive* or *negative*; here, positive means that a movie was rated with more than six stars on IMDb, and negative means that a movie was rated with fewer than five stars on IMDb. In the following sections, we will learn how to extract meaningful information from a subset of these movie reviews to build a machine learning model that can predict whether a certain reviewer liked or disliked a movie.

A compressed archive of the movie review dataset (84.1 MB) can be downloaded from `http://ai.stanford.edu/~amaas/data/sentiment/` as a gzip-compressed tarball archive:

- If you are working with Linux or Mac OS X, you can open a new terminal window, use `cd` to go into the `download` directory, and execute `tar -zxf aclImdb_v1.tar.gz` to decompress the dataset

- If you are working with Windows, you can download a free archiver such as 7-Zip (`http://www.7-zip.org`) to extract the files from the download archive

Having successfully extracted the dataset, we will now assemble the individual text documents from the decompressed download archive into a single CSV file. In the following code section, we will be reading the movie reviews into a pandas `DataFrame` object, which can take up to 10 minutes on a standard desktop computer. To visualize the progress and estimated time until completion, we will use the **PyPrind** (Python Progress Indicator, `https://pypi.python.org/pypi/PyPrind/`) package that I developed several years ago for such purposes. PyPrind can be installed by executing the command: `pip install pyprind`.

```
>>> import pyprind
>>> import pandas as pd
>>> import os
>>> pbar = pyprind.ProgBar(50000)
>>> labels = {'pos':1, 'neg':0}
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path ='./aclImdb/%s/%s' % (s, l)
...         for file in os.listdir(path):
```

```
...                with open(os.path.join(path, file), 'r') as infile:...
txt = infile.read()
...                df = df.append([[txt, labels[l]]], ignore_index=True)
...                pbar.update()
>>> df.columns = ['review', 'sentiment']
0%                             100%
[##############################] | ETA[sec]: 0.000
Total time elapsed: 725.001 sec
```

Executing the preceding code, we first initialized a new progress bar object `pbar` with 50,000 iterations, which is the number of documents we were going to read in. Using the nested `for` loops, we iterated over the `train` and `test` subdirectories in the main `aclImdb` directory and read the individual text files from the `pos` and `neg` subdirectories that we eventually appended to the `DataFrame df` — together with an integer class label (`1` = positive and `0` = negative).

Since the class labels in the assembled dataset are sorted, we will now shuffle `DataFrame` using the `permutation` function from the `np.random` submodule — this will be useful to split the dataset into training and test sets in later sections when we will stream the data from our local drive directly. For our own convenience, we will also store the assembled and shuffled movie review dataset as a CSV file:

```
>>> import numpy as np
>>> np.random.seed(0)
>>> df = df.reindex(np.random.permutation(df.index))
>>> df.to_csv('./movie_data.csv', index=False)
```

Since we are going to use this dataset later in this chapter, let us quickly confirm that we successfully saved the data in the right format by reading in the CSV and printing an excerpt of the first three samples:

```
>>> df = pd.read_csv('./movie_data.csv')
>>> df.head(3)
```

If you are running the code examples in IPython Notebook, you should now see the first three samples of the dataset, as shown in the following table:

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Introducing the bag-of-words model

We remember from *Chapter 4*, *Building Good Training Sets – Data Preprocessing,* that we have to convert categorical data, such as text or words, into a numerical form before we can pass it on to a machine learning algorithm. In this section, we will introduce the **bag-of-words** model that allows us to represent text as numerical feature vectors. The idea behind the bag-of-words model is quite simple and can be summarized as follows:

1. We create a **vocabulary** of unique **tokens**—for example, words—from the entire set of documents.

2. We construct a feature vector from each document that contains the counts of how often each word occurs in the particular document.

Since the unique words in each document represent only a small subset of all the words in the bag-of-words vocabulary, the feature vectors will consist of mostly zeros, which is why we call them **sparse**. Do not worry if this sounds too abstract; in the following subsections, we will walk through the process of creating a simple bag-of-words model step-by-step.

Transforming words into feature vectors

To construct a bag-of-words model based on the word counts in the respective documents, we can use the `CountVectorizer` class implemented in scikit-learn. As we will see in the following code section, the `CountVectorizer` class takes an array of text data, which can be documents or just sentences, and constructs the bag-of-words model for us:

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array([
...         'The sun is shining',
...         'The weather is sweet',
...         'The sun is shining and the weather is sweet'])
>>> bag = count.fit_transform(docs)
```

By calling the `fit_transform` method on `CountVectorizer`, we just constructed the vocabulary of the bag-of-words model and transformed the following three sentences into sparse feature vectors:

1. The sun is shining

2. The weather is sweet

3. The sun is shining and the weather is sweet

Now let us print the contents of the vocabulary to get a better understanding of the underlying concepts:

```
>>> print(count.vocabulary_)
{'the': 5, 'shining': 2, 'weather': 6, 'sun': 3, 'is': 1, 'sweet': 4,
'and': 0}
```

As we can see from executing the preceding command, the vocabulary is stored in a Python dictionary, which maps the unique words that are mapped to integer indices. Next let us print the feature vectors that we just created:

```
>>> print(bag.toarray())
[[0 1 1 1 0 1 0]
 [0 1 0 0 1 1 1]
 [1 2 1 1 1 2 1]]
```

Each index position in the feature vectors shown here corresponds to the integer values that are stored as dictionary items in the `CountVectorizer` vocabulary. For example, the first feature at index position `0` resembles the count of the word `and`, which only occurs in the last document, and the word `is` at index position `1` (the 2nd feature in the document vectors) occurs in all three sentences. Those values in the feature vectors are also called the **raw term frequencies**: *tf (t,d)* — the number of times a term *t* occurs in a document *d*.

> The sequence of items in the bag-of-words model that we just created is also called the **1-gram** or **unigram** model — each item or token in the vocabulary represents a single word. More generally, the contiguous sequences of items in NLP — words, letters, or symbols — is also called an **n-gram**. The choice of the number *n* in the n-gram model depends on the particular application; for example, a study by Kanaris et al. revealed that n-grams of size 3 and 4 yield good performances in anti-spam filtering of e-mail messages (Ioannis Kanaris, Konstantinos Kanaris, Ioannis Houvardas, and Efstathios Stamatatos. *Words vs Character N-Grams for Anti-Spam Filtering*. International Journal on Artificial Intelligence Tools, 16(06):1047–1067, 2007). To summarize the concept of the n-gram representation, the 1-gram and 2-gram representations of our first document "the sun is shining" would be constructed as follows:
>
> - **1-gram**: "the", "sun", "is", "shining"
> - **2-gram**: "the sun", "sun is", "is shining"
>
> The `CountVectorizer` class in scikit-learn allows us to use different n-gram models via its `ngram_range` parameter. While a 1-gram representation is used by default, we could switch to a 2-gram representation by initializing a new `CountVectorizer` instance with `ngram_range=(2,2)`.

Assessing word relevancy via term frequency-inverse document frequency

When we are analyzing text data, we often encounter words that occur across multiple documents from both classes. Those frequently occurring words typically don't contain useful or discriminatory information. In this subsection, we will learn about a useful technique called **term frequency-inverse document frequency** (**tf-idf**) that can be used to downweight those frequently occurring words in the feature vectors. The tf-idf can be defined as the product of the **term frequency** and the **inverse document frequency**:

$$\text{tf-idf}(t,d) = tf(t,d) \times \text{idf}(t,d)$$

Here the *tf(t, d)* is the term frequency that we introduced in the previous section, and the inverse document frequency *idf(t, d)* can be calculated as:

$$\text{idf}(t,d) = log \frac{n_d}{1 + \text{df}(d,t)},$$

where n_d is the total number of documents, and *df(d, t)* is the number of documents *d* that contain the term *t*. Note that adding the constant 1 to the denominator is optional and serves the purpose of assigning a non-zero value to terms that occur in all training samples; the log is used to ensure that low document frequencies are not given too much weight.

Scikit-learn implements yet another transformer, the `TfidfTransformer`, that takes the raw term frequencies from `CountVectorizer` as input and transforms them into tf-idfs:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf = TfidfTransformer()
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
[[ 0.     0.43   0.56   0.56   0.     0.43   0.   ]
 [ 0.     0.43   0.     0.     0.56   0.43   0.56]
 [ 0.4    0.48   0.31   0.31   0.31   0.48   0.31]]
```

As we saw in the previous subsection, the word `is` had the largest term frequency in the 3rd document, being the most frequently occurring word. However, after transforming the same feature vector into tf-idfs, we see that the word `is` is now associated with a relatively small tf-idf (`0.31`) in document 3 since it is also contained in documents 1 and 2 and thus is unlikely to contain any useful, discriminatory information.

However, if we'd manually calculated the tf-idfs of the individual terms in our feature vectors, we'd have noticed that the `TfidfTransformer` calculates the tf-idfs slightly differently compared to the *standard* textbook equations that we defined earlier. The equations for the idf and tf-idf that were implemented in scikit-learn are:

$$\text{idf}(t,d) = log\frac{1+n_d}{1+\text{df}(d,t)}$$

The tf-idf equation that was implemented in scikit-learn is as follows:

$$\text{tf-idf}(t,d) = tf(t,d) \times (\text{idf}(t,d)+1)$$

While it is also more typical to normalize the raw term frequencies before calculating the tf-idfs, the `TfidfTransformer` normalizes the tf-idfs directly. By default (`norm='l2'`), scikit-learn's `TfidfTransformer` applies the L2-normalization, which returns a vector of length 1 by dividing an un-normalized feature vector v by its L2-norm:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}} = \frac{v}{\left(\sum_{i=1}^{n} v_i^2\right)^{1/2}}$$

To make sure that we understand how `TfidfTransformer` works, let us walk through an example and calculate the tf-idf of the word `is` in the 3rd document.

The word `is` has a term frequency of 2 (tf = 2) in document 3, and the document frequency of this term is 3 since the term `is` occurs in all three documents (df = 3). Thus, we can calculate the idf as follows:

$$idf(\text{"is"}, d3) = log\frac{1+3}{1+3} = 0$$

Now in order to calculate the tf-idf, we simply need to add 1 to the inverse document frequency and multiply it by the term frequency:

$$\text{tf-idf}\left("is",\text{d}3\right)=2\times\left(0+1\right)=2$$

If we repeated these calculations for all terms in the 3rd document, we'd obtain the following tf-idf vectors: [1.69, 2.00, 1.29, 1.29, 1.29, 2.00, and 1.29]. However, we notice that the values in this feature vector are different from the values that we obtained from the `TfidfTransformer` that we used previously. The final step that we are missing in this tf-idf calculation is the L2-normalization, which can be applied as follows:

$$\text{tf-idf}\left("is",\text{d}3\right)=0.48$$

As we can see, the results now match the results returned by scikit-learn's `TfidfTransformer`. Since we now understand how tf-idfs are calculated, let us proceed to the next sections and apply those concepts to the movie review dataset.

Cleaning text data

In the previous subsections, we learned about the bag-of-words model, term frequencies, and tf-idfs. However, the first important step—before we build our bag-of-words model—is to clean the text data by stripping it of all unwanted characters. To illustrate why this is important, let us display the last 50 characters from the first document in the reshuffled movie review dataset:

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

As we can see here, the text contains HTML markup as well as punctuation and other non-letter characters. While HTML markup does not contain much useful semantics, punctuation marks can represent useful, additional information in certain NLP contexts. However, for simplicity, we will now remove all punctuation marks but only keep **emoticon** characters such as ":)" since those are certainly useful for sentiment analysis. To accomplish this task, we will use Python's **regular expression** (**regex**) library, re, as shown here:

```
>>> import re
>>> def preprocessor(text):
```

```
...          text = re.sub('<[^>]*>', '', text)
...          emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)
...          text = re.sub('[\W]+', ' ', text.lower()) + \
                      ' '.join(emoticons).replace('-', '')
...          return text
```

Via the first regex `<[^>]*>` in the preceding code section, we tried to remove the entire HTML markup that was contained in the movie reviews. Although many programmers generally advise against the use of regex to parse HTML, this regex should be sufficient to *clean* this particular dataset. After we removed the HTML markup, we used a slightly more complex regex to find emoticons, which we temporarily stored as `emoticons`. Next we removed all non-word characters from the text via the regex `[\W]+`, converted the text into lowercase characters, and eventually added the temporarily stored `emoticons` to the end of the processed document string. Additionally, we removed the *nose* character (-) from the emoticons for consistency.

> Although regular expressions offer an efficient and convenient approach to searching for characters in a string, they also come with a steep learning curve. Unfortunately, an in-depth discussion of regular expressions is beyond the scope of this book. However, you can find a great tutorial on the Google Developers portal at `https://developers.google.com/edu/python/regular-expressions` or check out the official documentation of Python's re module at `https://docs.python.org/3.4/library/re.html`.

Although the addition of the emoticon characters to the end of the cleaned document strings may not look like the most elegant approach, the order of the words doesn't matter in our bag-of-words model if our vocabulary only consists of 1-word tokens. But before we talk more about splitting documents into individual terms, words, or tokens, let us confirm that our preprocessor works correctly:

```
>>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

Lastly, since we will make use of the *cleaned* text data over and over again during the next sections, let us now apply our `preprocessor` function to all movie reviews in our `DataFrame`:

```
>>> df['review'] = df['review'].apply(preprocessor)
```

Processing documents into tokens

Having successfully prepared the movie review dataset, we now need to think about how to split the text corpora into individual elements. One way to *tokenize* documents is to split them into individual words by splitting the cleaned document at its whitespace characters:

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

In the context of tokenization, another useful technique is **word stemming**, which is the process of transforming a word into its root form that allows us to map related words to the same stem. The original stemming algorithm was developed by Martin F. Porter in 1979 and is hence known as the **Porter stemmer** algorithm (Martin F. Porter. *An algorithm for suffix stripping*. Program: electronic library and information systems, 14(3):130–137, 1980). The Natural Language Toolkit for Python (NLTK, http://www.nltk.org) implements the Porter stemming algorithm, which we will use in the following code section. In order to install the NLTK, you can simply execute pip install nltk.

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

> Although NLTK is not the focus of the chapter, I highly recommend you to visit the NLTK website as well as the official NLTK book, which is freely available at http://www.nltk.org/book/, if you are interested in more advanced applications in NLP.

Using PorterStemmer from the nltk package, we modified our tokenizer function to reduce words to their root form, which was illustrated by the previous simple example where the word running was stemmed to its root form run.

> The Porter stemming algorithm is probably the oldest and simplest stemming algorithm. Other popular stemming algorithms include the newer **Snowball stemmer** (Porter2 or "English" stemmer) or the **Lancaster stemmer** (Paice-Husk stemmer), which is faster but also more aggressive than the Porter stemmer. Those alternative stemming algorithms are also available through the NLTK package (`http://www.nltk.org/api/nltk.stem.html`).
>
> While stemming can create non-real words, such as `thu`, (from `thus`) as shown in the previous example, a technique called **lemmatization** aims to obtain the canonical (grammatically correct) forms of individual words—the so-called **lemmas**. However, lemmatization is computationally more difficult and expensive compared to stemming and, in practice, it has been observed that stemming and lemmatization have little impact on the performance of text classification (Michal Toman, Roman Tesar, and Karel Jezek. *Influence of word normalization on text classification*. Proceedings of InSciT, pages 354–358, 2006).

Before we jump into the next section where we will train a machine learning model using the bag-of-words model, let us briefly talk about another useful topic called **stop-word removal**. Stop-words are simply those words that are extremely common in all sorts of texts and likely bear no (or only little) useful information that can be used to distinguish between different classes of documents. Examples of stop-words are *is*, *and*, *has*, and the like. Removing stop-words can be useful if we are working with raw or normalized term frequencies rather than tf-idfs, which are already downweighting frequently occurring words.

In order to remove stop-words from the movie reviews, we will use the set of 127 English stop-words that is available from the NLTK library, which can be obtained by calling the `nltk.download` function:

```
>>> import nltk
>>> nltk.download('stopwords')
```

After we have downloaded the stop-words set, we can load and apply the English stop-word set as follows:

```
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> [w for w in tokenizer_porter('a runner likes running and runs a
lot')[-10:] if w not in stop]

['runner', 'like', 'run', 'run', 'lot']
```

Training a logistic regression model for document classification

In this section, we will train a logistic regression model to classify the movie reviews into positive and negative reviews. First, we will divide the `DataFrame` of cleaned text documents into 25,000 documents for training and 25,000 documents for testing:

```
>>> X_train = df.loc[:25000, 'review'].values
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = df.loc[25000:, 'review'].values
>>> y_test = df.loc[25000:, 'sentiment'].values
```

Next we will use a `GridSearchCV` object to find the optimal set of parameters for our logistic regression model using 5-fold stratified cross-validation:

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> tfidf = TfidfVectorizer(strip_accents=None,
...                         lowercase=False,
...                         preprocessor=None)
>>> param_grid = [{'vect__ngram_range': [(1,1)],
...                'vect__stop_words': [stop, None],
...                'vect__tokenizer': [tokenizer,
...                                    tokenizer_porter],
...                'clf__penalty': ['l1', 'l2'],
...                'clf__C': [1.0, 10.0, 100.0]},
...               {'vect__ngram_range': [(1,1)],
...                'vect__stop_words': [stop, None],
...                'vect__tokenizer': [tokenizer,
...                                    tokenizer_porter],
...                'vect__use_idf':[False],
...                'vect__norm':[None],
...                'clf__penalty': ['l1', 'l2'],
...                'clf__C': [1.0, 10.0, 100.0]}
...              ]
>>> lr_tfidf = Pipeline([('vect', tfidf),
...                     ('clf',
...                      LogisticRegression(random_state=0))])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
...                           scoring='accuracy',
...                           cv=5, verbose=1,
...                           n_jobs=-1)
>>> gs_lr_tfidf.fit(X_train, y_train)
```

When we initialized the `GridSearchCV` object and its parameter grid using the preceding code, we restricted ourselves to a limited number of parameter combinations since the number of feature vectors, as well as the large vocabulary, can make the grid search computationally quite expensive; using a standard Desktop computer, our grid search may take up to 40 minutes to complete.

In the previous code example, we replaced the `CountVectorizer` and `TfidfTransformer` from the previous subsection with the `TfidfVectorizer`, which combines the latter transformer objects. Our `param_grid` consisted of two parameter dictionaries. In the first dictionary, we used the `TfidfVectorizer` with its default settings (`use_idf=True`, `smooth_idf=True`, and `norm='l2'`) to calculate the tf-idfs; in the second dictionary, we set those parameters to `use_idf=False`, `smooth_idf=False`, and `norm=None` in order to train a model based on raw term frequencies. Furthermore, for the logistic regression classifier itself, we trained models using L2 and L1 regularization via the penalty parameter and compared different regularization strengths by defining a range of values for the inverse-regularization parameter `C`.

After the grid search has finished, we can print the best parameter set:

```
>>> print('Best parameter set: %s ' % gs_lr_tfidf.best_params_)
Best parameter set: {'clf__C': 10.0, 'vect__stop_words': None,
'clf__penalty': 'l2', 'vect__tokenizer': <function tokenizer at
0x7f6c704948c8>, 'vect__ngram_range': (1, 1)}
```

As we can see here, we obtained the best grid search results using the regular tokenizer without Porter stemming, no stop-word library, and tf-idfs in combination with a logistic regression classifier that uses L2 regularization with the regularization strength `C=10.0`.

Using the best model from this grid search, let us print the average 5-fold cross-validation accuracy score on the training set and the classification accuracy on the test dataset:

```
>>> print('CV Accuracy: %.3f'
...       % gs_lr_tfidf.best_score_)
CV Accuracy: 0.897
>>> clf = gs_lr_tfidf.best_estimator_
>>> print('Test Accuracy: %.3f'
...      % clf.score(X_test, y_test))
Test Accuracy: 0.899
```

The results reveal that our machine learning model can predict whether a movie review is positive or negative with 90 percent accuracy.

> A still very popular classifier for text classification is the Naïve Bayes classifier, which gained popularity in applications of e-mail spam filtering. Naïve Bayes classifiers are easy to implement, computationally efficient, and tend to perform particularly well on relatively small datasets compared to other algorithms. Although we don't discuss Naïve Bayes classifiers in this book, the interested reader can find my article about Naïve Text classification that I made freely available on arXiv (S. Raschka. *Naive Bayes and Text Classification I - introduction and Theory*. Computing Research Repository (CoRR), abs/1410.5329, 2014. `http://arxiv.org/pdf/1410.5329v3.pdf`).

Working with bigger data – online algorithms and out-of-core learning

If you executed the code examples in the previous section, you may have noticed that it could be computationally quite expensive to construct the feature vectors for the 50,000 movie review dataset during grid search. In many real-world applications it is not uncommon to work with even larger datasets that may even exceed our computer's memory. Since not everyone has access to supercomputer facilities, we will now apply a technique called out-of-core learning that allows us to work with such large datasets.

Back in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, we introduced the concept of **stochastic gradient descent**, which is an optimization algorithm that updates the model's weights using one sample at a time. In this section, we will make use of the `partial_fit` function of the `SGDClassifier` in scikit-learn to stream the documents directly from our local drive and train a logistic regression model using small minibatches of documents.

First, we define a `tokenizer` function that cleans the unprocessed text data from our `movie_data.csv` file that we constructed in the beginning of this chapter and separates it into word tokens while removing stop words.

```
>>> import numpy as np
>>> import re
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            text.lower())
...     text = re.sub('[\W]+', ' ', text.lower()) \
```

```
...             + ' '.join(emoticons).replace('-', '')
...         tokenized = [w for w in text.split() if w not in stop]
...         return tokenized
```

Next we define a generator function, `stream_docs`, that reads in and returns one document at a time:

```
>>> def stream_docs(path):
...     with open(path, 'r', encoding='utf-8') as csv:
...         next(csv) # skip header
...         for line in csv:
...             text, label = line[:-3], int(line[-2])
...             yield text, label
```

To verify that our `stream_docs` function works correctly, let us read in the first document from the `movie_data.csv` file, which should return a tuple consisting of the review text as well as the corresponding class label:

```
>>> next(stream_docs(path='./movie_data.csv'))
('"In 1974, the teenager Martha Moxley ... ',1)
```

We will now define a function, `get_minibatch`, that will take a document stream from the `stream_docs` function and return a particular number of documents specified by the `size` parameter:

```
>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...         try:
...             for _ in range(size):
...                 text, label = next(doc_stream)
...                 docs.append(text)
...                 y.append(label)
...         except StopIteration:
...             return None, None
...         return docs, y
```

Unfortunately, we can't use the `CountVectorizer` for out-of-core learning since it requires holding the complete vocabulary in memory. Also, the `TfidfVectorizer` needs to keep the all feature vectors of the training dataset in memory to calculate the inverse document frequencies. However, another useful vectorizer for text processing implemented in scikit-learn is `HashingVectorizer`. `HashingVectorizer` is data-independent and makes use of the Hashing trick via the 32-bit MurmurHash3 algorithm by Austin Appleby (`https://sites.google.com/site/murmurhash/`).

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier
```

```
>>> vect = HashingVectorizer(decode_error='ignore',
...                          n_features=2**21,
...                          preprocessor=None,
...                          tokenizer=tokenizer)
>>> clf = SGDClassifier(loss='log', random_state=1, n_iter=1)
>>> doc_stream = stream_docs(path='./movie_data.csv')
```

Using the preceding code, we initialized HashingVectorizer with our tokenizer function and set the number of features to 2^{21}. Furthermore, we reinitialized a logistic regression classifier by setting the loss parameter of the SGDClassifier to log—note that, by choosing a large number of features in the HashingVectorizer, we reduce the chance to cause hash collisions but we also increase the number of coefficients in our logistic regression model.

Now comes the really interesting part. Having set up all the complementary functions, we can now start the out-of-core learning using the following code:

```
>>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>>> classes = np.array([0, 1])
>>> for _ in range(45):
...     X_train, y_train = get_minibatch(doc_stream, size=1000)
...     if not X_train:
...         break
...     X_train = vect.transform(X_train)
...     clf.partial_fit(X_train, y_train, classes=classes)
...     pbar.update()
0%                         100%
[#############################] | ETA[sec]: 0.000
Total time elapsed: 50.063 sec
```

Again, we made use of the PyPrind package in order to estimate the progress of our learning algorithm. We initialized the progress bar object with 45 iterations and, in the following for loop, we iterated over 45 minibatches of documents where each minibatch consists of 1,000 documents each.

Having completed the incremental learning process, we will use the last 5,000 documents to evaluate the performance of our model:

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>>> X_test = vect.transform(X_test)
>>> print('Accuracy: %.3f' % clf.score(X_test, y_test))
Accuracy: 0.868
```

As we can see, the accuracy of the model is 87 percent, slightly below the accuracy that we achieved in the previous section using the grid search for hyperparameter tuning. However, out-of-core learning is very memory-efficient and took less than a minute to complete. Finally, we can use the last 5,000 documents to update our model:

```
>>> clf = clf.partial_fit(X_test, y_test)
```

If you are planning to continue directly with *Chapter 9*, *Embedding a Machine Learning Model into a Web Application*, I recommend you to keep the current Python session open. In the next chapter, we will use the model that we just trained to learn how to save it to disk for later use and embed it into a web application.

> Although the bag-of-words model is still the most commonly used model for text classification, it does not consider sentence structure and grammar. A popular extension of the bag-of-words model is **Latent Dirichlet allocation**, which is a topic model that considers the latent semantics of words (D. M. Blei, A. Y. Ng, and M. I. Jordan. *Latent Dirichlet allocation*. The Journal of machine Learning research, 3:993–1022, 2003).
>
> A more modern alternative to the bag-of-words model is **word2vec**, an algorithm that Google released in 2013 (T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. arXiv preprint arXiv:1301.3781, 2013). The word2vec algorithm is an unsupervised learning algorithm based on neural networks that attempts to automatically learn the relationship between words. The idea behind word2vec is to put words that have similar meanings into similar clusters; via clever vector-spacing, the model can reproduce certain words using simple vector math, for example, *king – man + woman = queen*.
>
> The original C-implementation, with useful links to the relevant papers and alternative implementations, can be found at `https://code.google.com/p/word2vec/`.

Summary

In this chapter, we learned how to use machine learning algorithms to classify text documents based on their polarity, which is a basic task in sentiment analysis in the field of natural language processing. Not only did we learn how to encode a document as a feature vector using the bag-of-words model, but we also learned how to weight the term frequency by relevance using term frequency-inverse document frequency.

Working with text data can be computationally quite expensive due to the large feature vectors that are created during this process; in the last section, we learned how to utilize out-of-core or incremental learning to train a machine learning algorithm without loading the whole dataset into a computer's memory.

In the next chapter, we will use our document classifier and learn how to embed it into a web application.

9
Embedding a Machine Learning Model into a Web Application

In the previous chapters, you learned about the many different machine learning concepts and algorithms that can help us with better and more efficient decision-making. However, machine learning techniques are not limited to offline applications and analyses, and they can be the predictive engine of your web services. For example, popular and useful applications of machine learning models in web applications include spam detection in submission forms, search engines, recommendation systems for media or shopping portals, and many more.

In this chapter, you will learn how to embed a machine learning model into a web application that can not only classify but also learn from data in real-time. The topics that we will cover are as follows:

- Saving the current state of a trained machine learning model
- Using SQLite databases for data storage
- Developing a web application using the popular Flask web framework
- Deploying a machine learning application to a public web server

Serializing fitted scikit-learn estimators

Training a machine learning model can be computationally quite expensive, as we have seen in *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*. Surely, we don't want to train our model every time we close our Python interpreter and want to make a new prediction or reload our web application? One option for **model persistence** is Python's in-built pickle module (`https://docs.python.org/3.4/library/pickle.html`), which allows us to serialize and de-serialize Python object structures to compact byte code, so that we can save our classifier in its current state and reload it if we want to classify new samples without needing to learn the model from the training data all over again. Before you execute the following code, please make sure that you have trained the out-of-core logistic regression model from the last section of *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*, and have it ready in your current Python session:

```
>>> import pickle
>>> import os
>>> dest = os.path.join('movieclassifier', 'pkl_objects')
>>> if not os.path.exists(dest):
...      os.makedirs(dest)
>>> pickle.dump(stop,
...          open(os.path.join(dest, 'stopwords.pkl'),'wb'),
...          protocol=4)
>>> pickle.dump(clf,
...          open(os.path.join(dest, 'classifier.pkl'), 'wb'),
...          protocol=4)
```

Using the preceding code, we created a `movieclassifier` directory where we will later store the files and data for our web application. Within this `movieclassifier` directory, we created a `pkl_objects` subdirectory to save the serialized Python objects to our local drive. Via pickle's `dump` method, we then serialized the trained logistic regression model as well as the stop word set from the NLTK library so that we don't have to install the NLTK vocabulary on our server. The `dump` method takes as its first argument the object that we want to pickle, and for the second argument we provided an open file object that the Python object will be written to. Via the `wb` argument inside the `open` function, we opened the file in binary mode for pickle, and we set `protocol=4` to choose the latest and most efficient pickle protocol that has been added to Python 3.4. (If you have problems using protocol 4, please check if you are using the latest Python 3 version install. Alternatively, you may consider choosing a lower protocol number.)

> Our logistic regression model contains several NumPy arrays, such as the weight vector, and a more efficient way to serialize NumPy arrays is to use the alternative joblib library. To ensure compatibility with the server environment that we will use in later sections, we will use the standard pickle approach. If you are interested, you can find more information about joblib at `https://pypi.python.org/pypi/joblib`.

We don't need to pickle the `HashingVectorizer`, since it does not need to be fitted. Instead, we can create a new Python script file, from which we can import the vectorizer into our current Python session. Now, copy the following code and save it as `vectorizer.py` in the `movieclassifier` directory:

```python
from sklearn.feature_extraction.text import HashingVectorizer
import re
import os
import pickle

cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(
                os.path.join(cur_dir,
                'pkl_objects',
                'stopwords.pkl'), 'rb'))

def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
                           text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) \
                   + ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized

vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
```

After we have pickled the Python objects and created the `vectorizer.py` file, it would now be a good idea to restart our Python interpreter or IPython Notebook kernel to test if we can deserialize the objects without error. However, please note that unpickling data from an untrusted source can be a potential security risk since the pickle module is not secure against malicious code. From your terminal, navigate to the `movieclassifier` directory, start a new Python session and execute the following code to verify that you can import the vectorizer and unpickle the classifier:

```
>>> import pickle
>>> import re
>>> import os
>>> from vectorizer import vect
>>> clf = pickle.load(open(
...         os.path.join('pkl_objects',
...                      'classifier.pkl'), 'rb'))
```

After we have successfully loaded the vectorizer and unpickled the classifier, we can now use these objects to pre-process document samples and make predictions about their sentiment:

```
>>> import numpy as np
>>> label = {0:'negative', 1:'positive'}
>>> example = ['I love this movie']
>>> X = vect.transform(example)
>>> print('Prediction: %s\nProbability: %.2f%%' %\
...        (label[clf.predict(X)[0]],
...         np.max(clf.predict_proba(X))*100))
Prediction: positive
Probability: 91.56%
```

Since our classifier returns the class labels as integers, we defined a simple Python dictionary to map those integers to their sentiment. We then used the `HashingVectorizer` to transform the simple example document into a word vector X. Finally, we used the `predict` method of the logistic regression classifier to predict the class label as well as the `predict_proba` method to return the corresponding probability of our prediction. Note that the `predict_proba` method call returns an array with a probability value for each unique class label. Since the class label with the largest probability corresponds to the class label that is returned by the `predict` call, we used the `np.max` function to return the probability of the predicted class.

Setting up a SQLite database for data storage

In this section, we will set up a simple **SQLite database** to collect optional feedback about the predictions from users of the web application. We can use this feedback to update our classification model. SQLite is an open source SQL database engine that doesn't require a separate server to operate, which makes it ideal for smaller projects and simple web applications. Essentially, a SQLite database can be understood as a single, self-contained database file that allows us to directly access storage files. Furthermore, SQLite doesn't require any system-specific configuration and is supported by all common operating systems. It has gained a reputation for being very reliable as it is used by popular companies, such as Google, Mozilla, Adobe, Apple, Microsoft, and many more. If you want to learn more about SQLite, I recommend you visit the official website at `http://www.sqlite.org`.

Fortunately, following Python's *batteries included* philosophy, there is already an API in the Python standard library, **sqlite3**, which allows us to work with SQLite databases (for more information about sqlite3, please visit `https://docs.python.org/3.4/library/sqlite3.html`).

By executing the following code, we will create a new SQLite database inside the `movieclassifier` directory and store two example movie reviews:
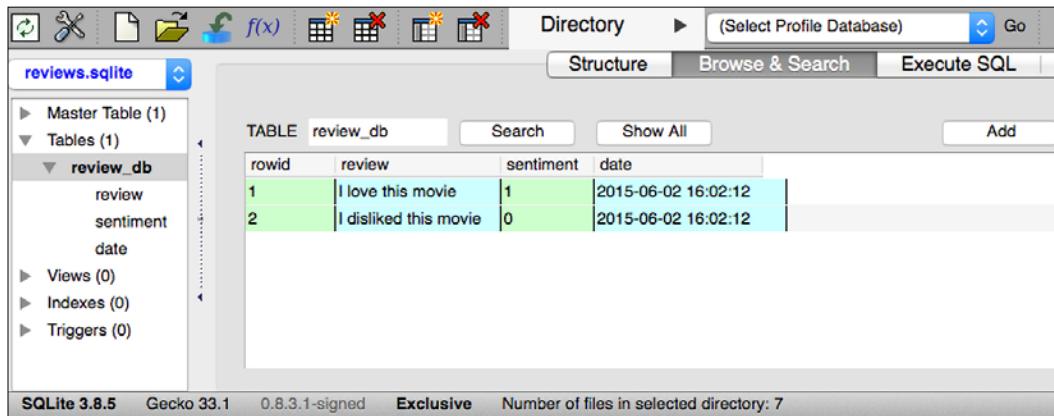
```
>>> import sqlite3
>>> import os
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute('CREATE TABLE review_db'\
...           ' (review TEXT, sentiment INTEGER, date TEXT)')
>>> example1 = 'I love this movie'
>>> c.execute("INSERT INTO review_db"\
...           " (review, sentiment, date) VALUES"\
...           " (?, ?, DATETIME('now'))", (example1, 1))
>>> example2 = 'I disliked this movie'
>>> c.execute("INSERT INTO review_db"\
...           " (review, sentiment, date) VALUES"\
...           " (?, ?, DATETIME('now'))", (example2, 0))
>>> conn.commit()
>>> conn.close()
```

Following the preceding code example, we created a connection (`conn`) to an SQLite database file by calling sqlite3's `connect` method, which created the new database file `reviews.sqlite` in the `movieclassifier` directory if it didn't already exist. Please note that SQLite doesn't implement a replace function for existing tables; you need to delete the database file manually from your file browser if you want to execute the code a second time. Next, we created a cursor via the `cursor` method, which allows us to traverse over the database records using the powerful SQL syntax. Via the first `execute` call, we then created a new database table, `review_db`. We used this to store and access database entries. Along with `review_db`, we also created three columns in this database table: `review`, `sentiment`, and `date`. We used these to store two example movie reviews and respective class labels (sentiments). Using the SQL command `DATETIME('now')`, we also added date-and timestamps to our entries. In addition to the timestamps, we used the question mark symbols (`?`) to pass the movie review texts (`example1` and `example2`) and the corresponding class labels (`1` and `0`) as positional arguments to the `execute` method as members of a tuple. Lastly, we called the `commit` method to save the changes that we made to the database and closed the connection via the `close` method.

To check if the entries have been stored in the database table correctly, we will now reopen the connection to the database and use the SQL `SELECT` command to fetch all rows in the database table that have been committed between the beginning of the year 2015 and today:

```
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute("SELECT * FROM review_db WHERE date"\
...      " BETWEEN '2015-01-01 00:00:00' AND DATETIME('now')")
>>> results = c.fetchall()
>>> conn.close()
>>> print(results)
[('I love this movie', 1, '2015-06-02 16:02:12'), ('I disliked this
movie', 0, '2015-06-02 16:02:12')]
```

Alternatively, we could also use the free Firefox browser plugin **SQLite Manager** (available at `https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/`), which offers a nice GUI interface for working with SQLite databases as shown in the following screenshot:

Developing a web application with Flask

After we have prepared the code to classify movie reviews in the previous subsection, let's discuss the basics of the Flask web framework to develop our web application. After Armin Ronacher's initial release of Flask in 2010, the framework has gained huge popularity over the years and examples of popular applications that make use of Flask include LinkedIn and Pinterest. Since Flask is written in Python, it provides us Python programmers with a convenient interface for embedding existing Python code such as our movie classifier.

> Flask is also known as a *microframework*, which means that its core is kept lean and simple but can be easily extended with other libraries. Although the learning curve of the lightweight Flask API is not nearly as steep as those of other popular Python web frameworks, such as Django, I encourage you to take a look at the official Flask documentation at `http://flask.pocoo.org/docs/0.10/` to learn more about its functionality.

If the Flask library is not already installed in your current Python environment, you can simply install it via pip from your terminal (at the time of writing, the latest stable release was Version 0.10.1):

```
pip install flask
```

Our first Flask web application

In this subsection, we will develop a very simple web application to become more familiar with the Flask API before we implement our movie classifier. First, we create a directory tree:

```
1st_flask_app_1/
    app.py
    templates/
        first_app.html
```

The app.py file will contain the main code that will be executed by the Python interpreter to run the Flask web application. The templates directory is the directory in which Flask will look for static HTML files for rendering in the web browser. Let's now take a look at the contents of app.py:

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('first_app.html')

if __name__ == '__main__':
    app.run()
```

In this case, we run our application as a single module, thus we initialized a new Flask instance with the argument __name__ to let Flask know that it can find the HTML template folder (templates) in the same directory where it is located. Next, we used the route decorator (@app.route('/')) to specify the URL that should trigger the execution of the index function. Here, our index function simply renders the HTML file first_app.html, which is located in the templates folder. Lastly, we used the run function to only run the application on the server when this script is directly executed by the Python interpreter, which we ensured using the if statement with __name__ == '__main__'.

Now, let's take a look at the contents of the `first_app.html` file. If you are not familiar with the HTML syntax yet, I recommend you visit `https://developer.mozilla.org/en-US/docs/Web/HTML` for useful tutorials for learning the basics of HTML.

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  </head>
  <body>
  <div>Hi, this is my first Flask web app!</div>
  </body>
</html>
```

Here, we have simply filled an empty HTML template file with a `div` element (a block level element) that contains the sentence: `Hi, this is my first Flask web app!`. Conveniently, Flask allows us to run our apps locally, which is useful for developing and testing web applications before we deploy them on a public web server. Now, let's start our web application by executing the command from the terminal inside the `1st_flask_app_1` directory:

```
python3 app.py
```

We should now see a line such as the following displayed in the terminal:

```
* Running on http://127.0.0.1:5000/
```
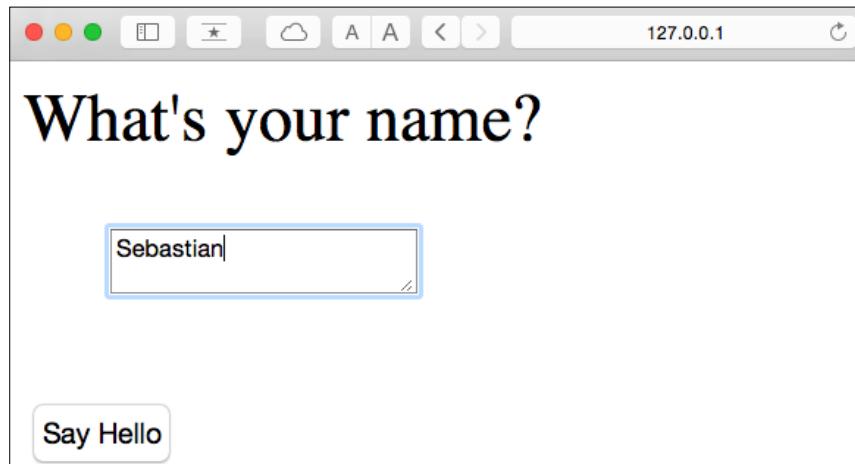
This line contains the address of our local server. We can now enter this address in our web browser to see the web application in action. If everything has executed correctly, we should now see a simple website with the content: **Hi, this is my first Flask web app!**.

Form validation and rendering

In this subsection, we will extend our simple Flask web application with HTML form elements to learn how to collect data from a user using the **WTForms** library (`https://wtforms.readthedocs.org/en/latest/`), which can be installed via pip:

```
pip install wtforms
```

This web app will prompt a user to type in his or her name into a text field, as shown in the following screenshot:

After the submission button (**Say Hello**) has been clicked and the form is validated, a new HTML page will be rendered to display the user's name.

The new directory structure that we need to set up for this application looks like this:

```
1st_flask_app_2/
    app.py
    static/
        style.css
    templates/
        _formhelpers.html
        first_app.html
        hello.html
```

The following are the contents of our modified `app.py` file:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
```

```
app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('',[validators.DataRequired()])

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)
```

Using `wtforms`, we extended the `index` function with a text field that we will embed in our start page using the `TextAreaField` class, which automatically checks whether a user has provided valid input text or not. Furthermore, we defined a new function, `hello`, which will render an HTML page `hello.html` if the form has been validated. Here, we used the POST method to transport the form data to the server in the message body. Finally, by setting the argument `debug=True` inside the `app.run` method, we further activated Flask's debugger. This is a useful feature for developing new web applications.

Now, we will implement a generic macro in the file `_formhelpers.html` via the **Jinja2** templating engine, which we will later import in our `first_app.html` file to render the text field:

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
  </dt>
{% endmacro %}
```

An in-depth discussion about the Jinja2 templating language is beyond the scope of this book. However, you can find a comprehensive documentation of the Jinja2 syntax at `http://jinja.pocoo.org`.

Next, we set up a simple **Cascading Style Sheets** (**CSS**) file, `style.css`, to demonstrate how the look and feel of HTML documents can be modified. We have to save the following CSS file, which will simply double the font size of our HTML body elements, in a subdirectory called `static`, which is the default directory where Flask looks for static files such as CSS. The code is as follows:

```
body {
  font-size: 2em;
}
```

The following are the contents of the modified `first_app.html` file that will now render a text form where a user can enter a name:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  <link rel="stylesheet" href="{{ url_for('static',
    filename='style.css') }}">
  </head>
  <body>

{% from "_formhelpers.html" import render_field %}

<div>What's your name?</div>
<form method=post action="/hello">
  <dl>
    {{ render_field(form.sayhello) }}
  </dl>
  <input type=submit value='Say Hello' name='submit_btn'>
</form>
  </body>
</html>
```

In the header section of `first_app.html`, we loaded the CSS file. It should now alter the size of all text elements in the HTML body. In the HTML body section, we imported the form macro from `_formhelpers.html` and we rendered the `sayhello` form that we specified in the `app.py` file. Furthermore, we added a button to the same form element so that a user can submit the text field entry.

Lastly, we create a `hello.html` file that will be rendered via the line return `render_template('hello.html', name=name)` inside the `hello` function, which we defined in the `app.py` script to display the text that a user submitted via the text field. The code is as follows:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  <link rel="stylesheet" href="{{ url_for('static',
    filename='style.css') }}">
  </head>
  <body>

  <div>Hello {{ name }}</div>
  </body>
</html>
```

Having set up our modified Flask web application, we can run it locally by executing the following command from the app's main directory and we can view the result in our web browser at `http://127.0.0.1:5000/`:
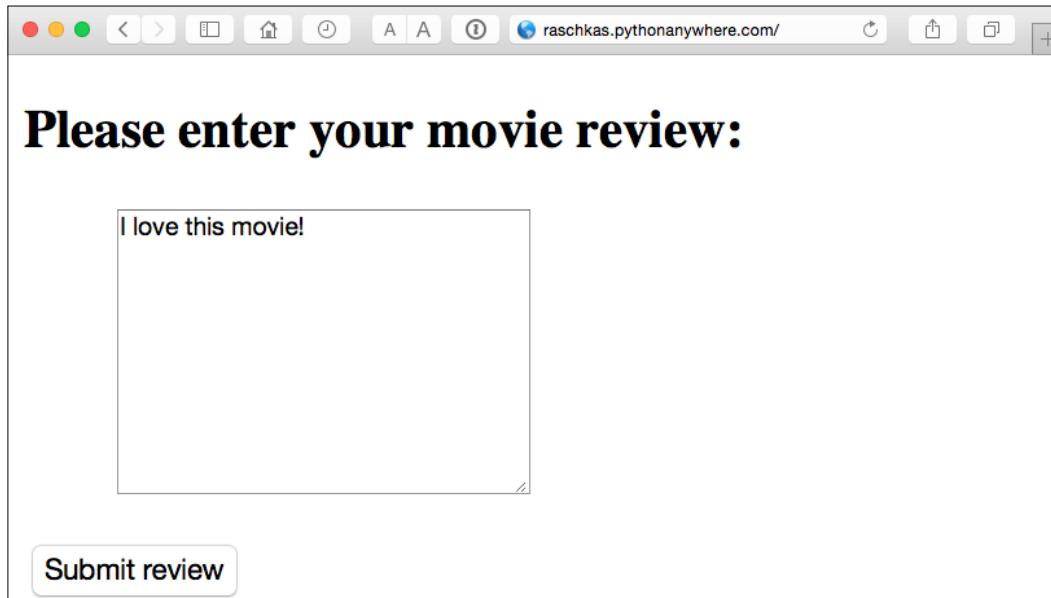
python3 app.py

> If you are new to web development, some of those concepts may seem very complicated at first sight. In that case, I encourage you to simply set up the preceding files in a directory on your hard drive and examine them closely. You will see that the Flask web framework is actually pretty straightforward and much simpler than it might initially appear! Also, for more help, don't forget to look at the excellent Flask documentation and examples at `http://flask.pocoo.org/docs/0.10/`.
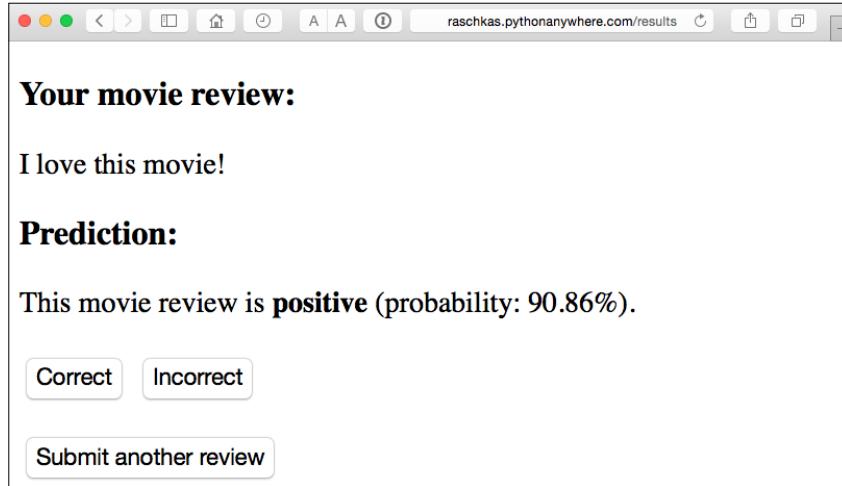
Turning the movie classifier into a web application
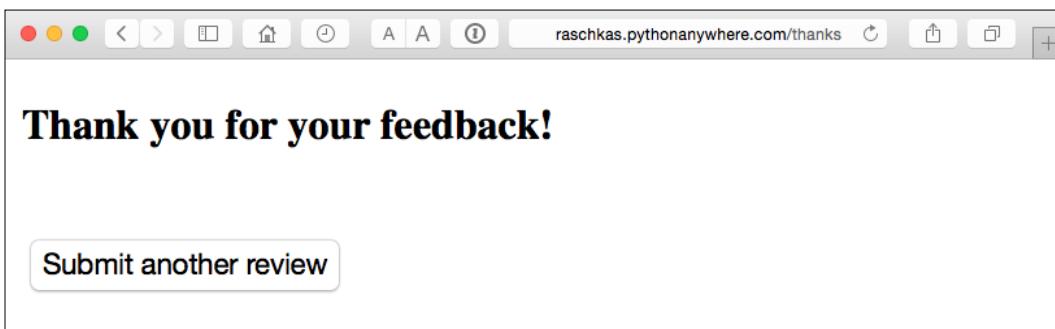
Now that we are somewhat familiar with the basics of Flask web development, let's advance to the next step and implement our movie classifier into a web application. In this section, we will develop a web application that will first prompt a user to enter a movie review, as shown in the following screenshot:

After the review has been submitted, the user will see a new page that shows the predicted class label and the probability of the prediction. Furthermore, the user will be able to provide feedback about this prediction by clicking on the **Correct** or **Incorrect** button, as shown in the following screenshot:
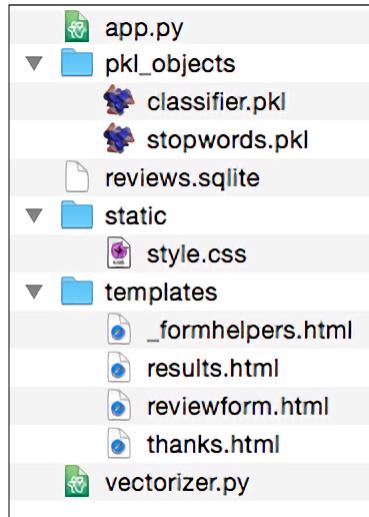
If a user clicked on either the **Correct** or **Incorrect** button, our classification model will be updated with respect to the user's feedback. Furthermore, we will also store the movie review text provided by the user as well as the suggested class label, which can be inferred from the button click, in a SQLite database for future reference. The third page that the user will see after clicking on one of the feedback buttons is a simple *thank you* screen with a **Submit another review** button that redirects the user back to the start page. This is shown in the following screenshot:

Before we take a closer look at the code implementation of this web application, I encourage you to take a look at the live demo that I uploaded at `http://raschkas.pythonanywhere.com` to get a better understanding of what we are trying to accomplish in this section.

To start with the big picture, let's take a look at the directory tree that we are going to create for this movie classification app, which is shown here:

In the previous section of this chapter, we already created the `vectorizer.py` file, the SQLite database `reviews.sqlite`, and the `pkl_objects` subdirectory with the pickled Python objects.

The `app.py` file in the main directory is the Python script that contains our Flask code, and we will use the `review.sqlite` database file (which we created earlier in this chapter) to store the movie reviews that are being submitted to our web app. The `templates` subdirectory contains the HTML templates that will be rendered by Flask and displayed in the browser, and the `static` subdirectory will contain a simple CSS file to adjust the look of the rendered HTML code.

Since the `app.py` file is rather long, we will conquer it in two steps. The first section of `app.py` imports the Python modules and objects that we are going to need, as well as the code to unpickle and set up our classification model:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np
```

```python
# import HashingVectorizer from local dir
from vectorizer import vect

app = Flask(__name__)

######## Preparing the Classifier
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                  'pkl_objects/classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'negative', 1: 'positive'}
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba =  clf.predict_proba(X).max()
    return label[y], proba

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO review_db (review, sentiment, date)"\
    " VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()
```

This first part of the `app.py` script should look very familiar to us by now. We simply imported the `HashingVectorizer` and unpickled the logistic regression classifier. Next, we defined a `classify` function to return the predicted class label as well as the corresponding probability prediction of a given text document. The `train` function can be used to update the classifier given that a document and a class label are provided. Using the `sqlite_entry` function, we can store a submitted movie review in our SQLite database along with its class label and timestamp for our personal records. Note that the `clf` object will be reset to its original, pickled state if we restart the web application. At the end of this chapter, you will learn how to use the data that we collect in the SQLite database to update the classifier permanently.

The concepts in the second part of the app.py script should also look quite familiar to us:

```
app = Flask(__name__)
class ReviewForm(Form):
    moviereview = TextAreaField('',
                               [validators.DataRequired(),
                                validators.length(min=15)])


@app.route('/')
def index():
    form = ReviewForm(request.form)
    return render_template('reviewform.html', form=form)

@app.route('/results', methods=['POST'])
def results():
    form = ReviewForm(request.form)
    if request.method == 'POST' and form.validate():
        review = request.form['moviereview']
        y, proba = classify(review)
        return render_template('results.html',
                               content=review,
                               prediction=y,
                               probability=round(proba*100, 2))
    return render_template('reviewform.html', form=form)

@app.route('/thanks', methods=['POST'])
def feedback():
    feedback = request.form['feedback_button']
    review = request.form['review']
    prediction = request.form['prediction']

    inv_label = {'negative': 0, 'positive': 1}
    y = inv_label[prediction]
    if feedback == 'Incorrect':
        y = int(not(y))
    train(review, y)
    sqlite_entry(db, review, y)
    return render_template('thanks.html')

if __name__ == '__main__':
    app.run(debug=True)
```

We defined a `ReviewForm` class that instantiates a `TextAreaField`, which will be rendered in the `reviewform.html` template file (the landing page of our web app). This, in turn, is rendered by the `index` function. With the `validators.length(min=15)` parameter, we require the user to enter a review that contains at least 15 characters. Inside the `results` function, we fetch the contents of the submitted web form and pass it on to our classifier to predict the sentiment of the movie classifier, which will then be displayed in the rendered `results.html` template.

The `feedback` function may look a little bit complicated at first glance. It essentially fetches the predicted class label from the `results.html` template if a user clicked on the **Correct** or **Incorrect** feedback button, and transforms the predicted sentiment back into an integer class label that will be used to update the classifier via the `train` function, which we implemented in the first section of the `app.py` script. Also, a new entry to the SQLite database will be made via the `sqlite_entry` function if feedback was provided, and eventually the `thanks.html` template will be rendered to thank the user for the feedback.

Next, let's take a look at the `reviewform.html` template, which constitutes the starting page of our application:

```
<!doctype html>
<html>
<head>
    <title>Movie Classification</title>
</head>
  <body>

<h2>Please enter your movie review:</h2>

{% from "_formhelpers.html" import render_field %}

<form method=post action="/results">
  <dl>
    {{ render_field(form.moviereview, cols='30', rows='10') }}
  </dl>
  <div>
    <input type=submit value='Submit review' name='submit_btn'>
  </div>
</form>

  </body>
</html>
```

Here, we simply imported the same _formhelpers.html template that we defined in the *Form validation and rendering* section earlier in this chapter. The render_field function of this macro is used to render a TextAreaField where a user can provide a movie review and submit it via the **Submit review** button displayed at the bottom of the page. This TextAreaField is 30 columns wide and 10 rows tall.

Our next template, results.html, looks a little bit more interesting:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
  <link rel="stylesheet" href="{{ url_for('static',
    filename='style.css') }}">
  </head>
  <body>

<h3>Your movie review:</h3>
<div>{{ content }}</div>

<h3>Prediction:</h3>
<div>This movie review is <strong>{{ prediction }}</strong>
  (probability: {{ probability }}%).</div>

<div class='button'>
  <form action="/thanks" method="post">
    <input type=submit value='Correct' name='feedback_button'>
    <input type=submit value='Incorrect' name='feedback_button'>
    <input type=hidden value='{{ prediction }}' name='prediction'>
    <input type=hidden value='{{ content }}' name='review'>
  </form>
</div>

<div class='button'>
  <form action="/">
    <input type=submit value='Submit another review'>
  </form>
</div>

  </body>
</html>
```

First, we inserted the submitted review as well as the results of the prediction in the corresponding fields {{ content }}, {{ prediction }}, and {{ probability }}. You may notice that we used the {{ content }} and {{ prediction }} placeholder variables a second time in the form that contains the **Correct** and **Incorrect** buttons. This is a workaround to POST those values back to the server to update the classifier and store the review in case the user clicks on one of those two buttons. Furthermore, we imported a CSS file (style.css) at the beginning of the results.html file. The setup of this file is quite simple; it limits the width of the contents of this web app to 600 pixels and moves the **Incorrect** and **Correct** buttons labeled with the div id button down by 20 pixels:

```
body{
  width:600px;
}
.button{
  padding-top: 20px;
}
```

This CSS file is merely a placeholder, so please feel free to adjust it to adjust the look and feel of the web app to your liking.

The last HTML file we will implement for our web application is the thanks.html template. As the name suggests, it simply provides a nice *thank you* message to the user after providing feedback via the **Correct** or **Incorrect** button. Furthermore, we put a **Submit another review** button at the bottom of this page, which will redirect the user to the starting page. The contents of the thanks.html file are as follows:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
</head>
  <body>

<h3>Thank you for your feedback!</h3>
<div id='button'>
  <form action="/">
    <input type=submit value='Submit another review'>
  </form>
</div>

  </body>
</html>
```

Now, it would be a good idea to start the web app locally from our terminal via the following command before we advance to the next subsection and deploy it on a public web server:

```
python3 app.py
```

After we have finished testing our app, we also shouldn't forget to remove the `debug=True` argument in the `app.run()` command of our `app.py` script.
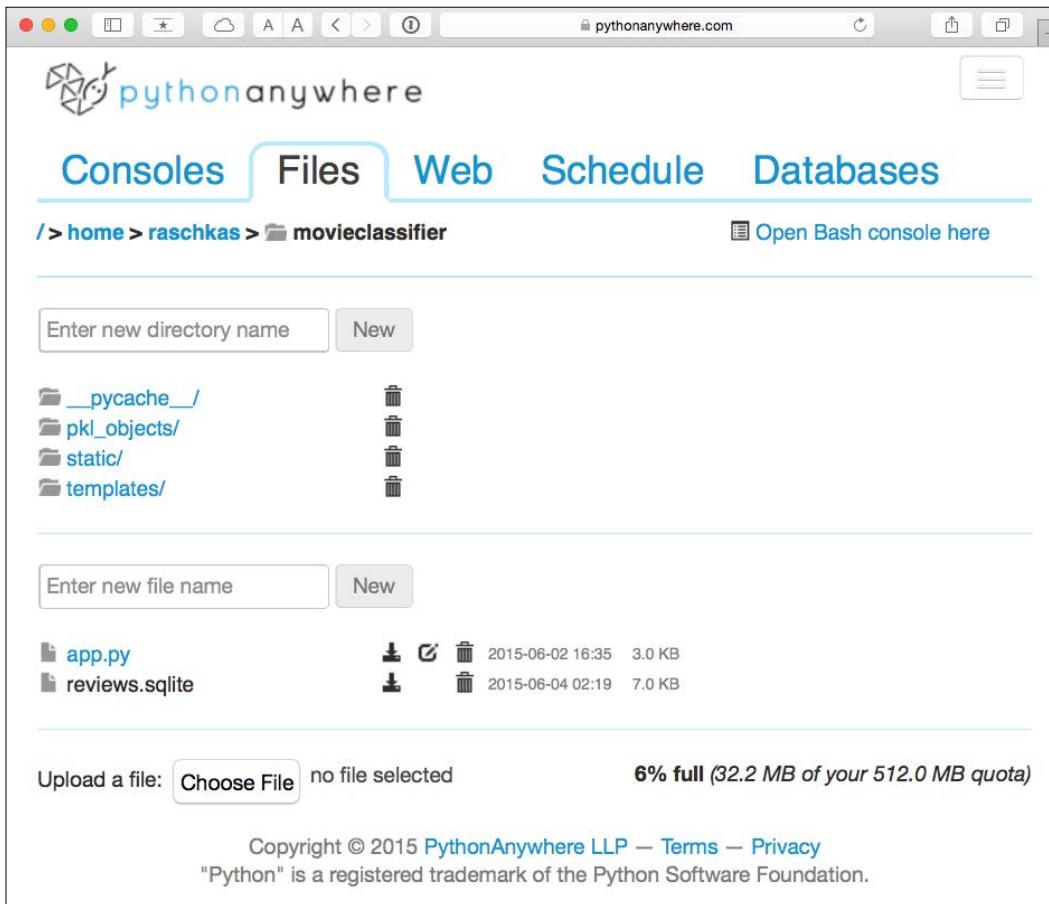
Deploying the web application to a public server

After we have tested the web application locally, we are now ready to deploy our web application onto a public web server. For this tutorial, we will be using the **PythonAnywhere** web hosting service, which specializes in the hosting of Python web applications and makes it extremely simple and hassle-free. Furthermore, PythonAnywhere offers a beginner account option that lets us run a single web application free of charge.

To create a new PythonAnywhere account, we visit the website at `https://www.pythonanywhere.com` and click on the **Pricing & signup** link that is located in the top-right corner. Next, we click on the **Create a Beginner account** button where we need to provide a username, password, and a valid e-mail address. After we have read and agreed to the terms and conditions, we should have a new account.

Unfortunately, the free beginner account doesn't allow us to access the remote server via the SSH protocol from our command-line terminal. Thus, we need to use the PythonAnywhere web interface to manage our web application. But before we can upload our local application files to the server, we need to create a new web application for our PythonAnywhere account. After we click on the **Dashboard** button in the top-right corner, we have access to the control panel shown at the top of the page. Next, we click on the **Web** tab that is now visible at the top of the page. We proceed by clicking on the **Add a new web app** button on the left, which lets us create a new Python 3.4 Flask web application that we name `movieclassifier`.

After creating a new application for our PythonAnywhere account, we head over to the **Files** tab to upload the files from our local `movieclassifier` directory using the PythonAnywhere web interface. After uploading the web application files that we created locally on our computer, we should have a `movieclassifier` directory in our PythonAnywhere account. It contains the same directories and files as our local `movieclassifier` directory has, as shown in the following screenshot:

Lastly, we head over to the **Web** tab one more time and click on the **Reload <username>.pythonanywhere.com** button to propagate the changes and refresh our web application. Finally, our web app should now be up and running and publicly available via the address `<username>.pythonanywhere.com`.

> Unfortunately, web servers can be quite sensitive to the tiniest problems in our web app. If you are experiencing problems with running the web application on PythonAnywhere and are receiving error messages in your browser, you can check the server and error logs which can be accessed from the **Web** tab in your PythonAnywhere account to better diagnose the problem.

Updating the movie review classifier

While our predictive model is updated on-the-fly whenever a user provides feedback about the classification, the updates to the `clf` object will be reset if the web server crashes or restarts. If we reload the web application, the `clf` object will be reinitialized from the `classifier.pkl` pickle file. One option to apply the updates permanently would be to pickle the `clf` object once again after each update. However, this would become computationally very inefficient with a growing number of users and could corrupt the pickle file if users provide feedback simultaneously. An alternative solution is to update the predictive model from the feedback data that is being collected in the SQLite database. One option would be to download the SQLite database from the PythonAnywhere server, update the `clf` object locally on our computer, and upload the new pickle file to PythonAnywhere. To update the classifier locally on our computer, we create an `update.py` script file in the `movieclassifier` directory with the following contents:

```python
import pickle
import sqlite3
import numpy as np
import os

# import HashingVectorizer from local dir
from vectorizer import vect

def update_model(db_path, model, batch_size=10000):

    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute('SELECT * from review_db')

    results = c.fetchmany(batch_size)
    while results:
        data = np.array(results)
        X = data[:, 0]
        y = data[:, 1].astype(int)

        classes = np.array([0, 1])
        X_train = vect.transform(X)
        model.partial_fit(X_train, y, classes=classes)
        results = c.fetchmany(batch_size)

    conn.close()
    return model
```

```
cur_dir = os.path.dirname(__file__)

clf = pickle.load(open(os.path.join(cur_dir,
                  'pkl_objects',
                  'classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

update_model(db_path=db, model=clf, batch_size=10000)

# Uncomment the following lines if you are sure that
# you want to update your classifier.pkl file
# permanently.

# pickle.dump(clf, open(os.path.join(cur_dir,
#             'pkl_objects', 'classifier.pkl'), 'wb')
#             , protocol=4)
```

The `update_model` function will fetch entries from the SQLite database in batches of 10,000 entries at a time unless the database contains fewer entries. Alternatively, we could also fetch one entry at a time by using `fetchone` instead of `fetchmany`, which would be computationally very inefficient. Using the alternative `fetchall` method could be a problem if we are working with large datasets that exceed the computer or server's memory capacity.

Now that we have created the `update.py` script, we could also upload it to the `movieclassifier` directory on PythonAnywhere and import the `update_model` function in the main application script `app.py` to update the classifier from the SQLite database every time we restart the web application. In order to do so, we just need to add a line of code to import the `update_model` function from the `update.py` script at the top of `app.py`:

```
# import update function from local dir
from update import update_model
```

We then need to call the `update_model` function in the main application body:

```
…
if __name__ == '__main__':
    clf = update_model(db_path="db", model=clf, batch_size=10000)
…
```

Summary

In this chapter, you learned about many useful and practical topics that extend our knowledge of machine learning theory. You learned how to serialize a model after training and how to load it for later use cases. Furthermore, we created a SQLite database for efficient data storage and created a web application that lets us make our movie classifier available to the outside world.

Throughout this book, we have really discussed a lot about machine learning concepts, best practices, and supervised models for classification. In the next chapter, we will take a look at another subcategory of supervised learning, regression analysis, which lets us predict outcome variables on a continuous scale, in contrast to the categorical class labels of the classification models that we have been working with so far.

10
Predicting Continuous Target Variables with Regression Analysis

Throughout the previous chapters, you learned a lot about the main concepts behind *supervised learning* and trained many different models for classification tasks to predict group memberships or categorical variables. In this chapter, we will take a dive into another subcategory of supervised learning: *regression analysis*.

Regression models are used to predict target variables on a *continuous* scale, which makes them attractive for addressing many questions in science as well as applications in industry, such as understanding relationships between variables, evaluating trends, or making forecasts. One example would be predicting the sales of a company in future months.

In this chapter, we will discuss the main concepts of regression models and cover the following topics:

- Exploring and visualizing datasets
- Looking at different approaches to implement linear regression models
- Training regression models that are robust to outliers
- Evaluating regression models and diagnosing common problems
- Fitting regression models to nonlinear data
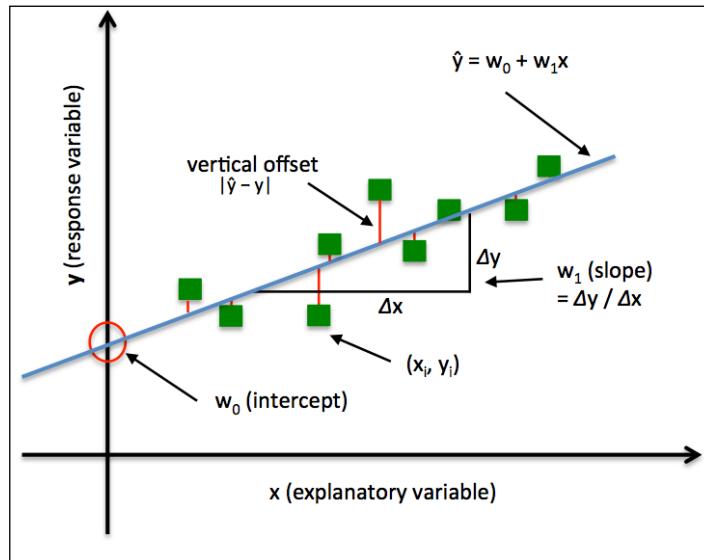
Introducing a simple linear regression model

The goal of simple (*univariate*) linear regression is to model the relationship between a single feature (explanatory variable x) and a continuous valued *response* (target variable y). The equation of a linear model with one explanatory variable is defined as follows:

$$y = w_0 + w_1 x$$

Here, the weight w_0 represents the y axis intercepts and w_1 is the coefficient of the explanatory variable. Our goal is to learn the weights of the linear equation to describe the relationship between the explanatory variable and the target variable, which can then be used to predict the responses of new explanatory variables that were not part of the training dataset.

Based on the linear equation that we defined previously, linear regression can be understood as finding the best-fitting straight line through the sample points, as shown in the following figure:

This best-fitting line is also called the **regression line**, and the vertical lines from the regression line to the sample points are the so-called **offsets** or **residuals**—the errors of our prediction.

The special case of one explanatory variable is also called **simple linear regression**, but of course we can also generalize the linear regression model to multiple explanatory variables. Hence, this process is called **multiple linear regression**:

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{i=0}^{m} w_i x_i = w^T x$$

Here, w_0 is the y axis intercept with $x_0 = 1$.

Exploring the Housing Dataset

Before we implement our first linear regression model, we will introduce a new dataset, the **Housing Dataset**, which contains information about houses in the suburbs of Boston collected by D. Harrison and D.L. Rubinfeld in 1978. The *Housing Dataset* has been made freely available and can be downloaded from the *UCI machine learning repository* at `https://archive.ics.uci.edu/ml/datasets/Housing`.

The features of the 506 samples may be summarized as shown in the excerpt of the dataset description:

- **CRIM**: This is the per capita crime rate by town
- **ZN**: This is the proportion of residential land zoned for lots larger than 25,000 sq.ft.
- **INDUS**: This is the proportion of non-retail business acres per town
- **CHAS**: This is the Charles River dummy variable (this is equal to 1 if tract bounds river; 0 otherwise)
- **NOX**: This is the nitric oxides concentration (parts per 10 million)
- **RM**: This is the average number of rooms per dwelling
- **AGE**: This is the proportion of owner-occupied units built prior to 1940
- **DIS**: This is the weighted distances to five Boston employment centers
- **RAD**: This is the index of accessibility to radial highways
- **TAX**: This is the full-value property-tax rate per $10,000
- **PTRATIO**: This is the pupil-teacher ratio by town
- **B**: This is calculated as *1000(Bk - 0.63)^2*, where Bk is the proportion of people of African American descent by town
- **LSTAT**: This is the percentage lower status of the population
- **MEDV**: This is the median value of owner-occupied homes in $1000s

For the rest of this chapter, we will regard the housing prices (MEDV) as our target variable—the variable that we want to predict using one or more of the 13 explanatory variables. Before we explore this dataset further, let's fetch it from the UCI repository into a pandas `DataFrame`:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/housing/housing.data',
...                  header=None, sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...               'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...               'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

To confirm that the dataset was loaded successfully, we displayed the first five lines of the dataset, as shown in the following screenshot:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

Visualizing the important characteristics of a dataset
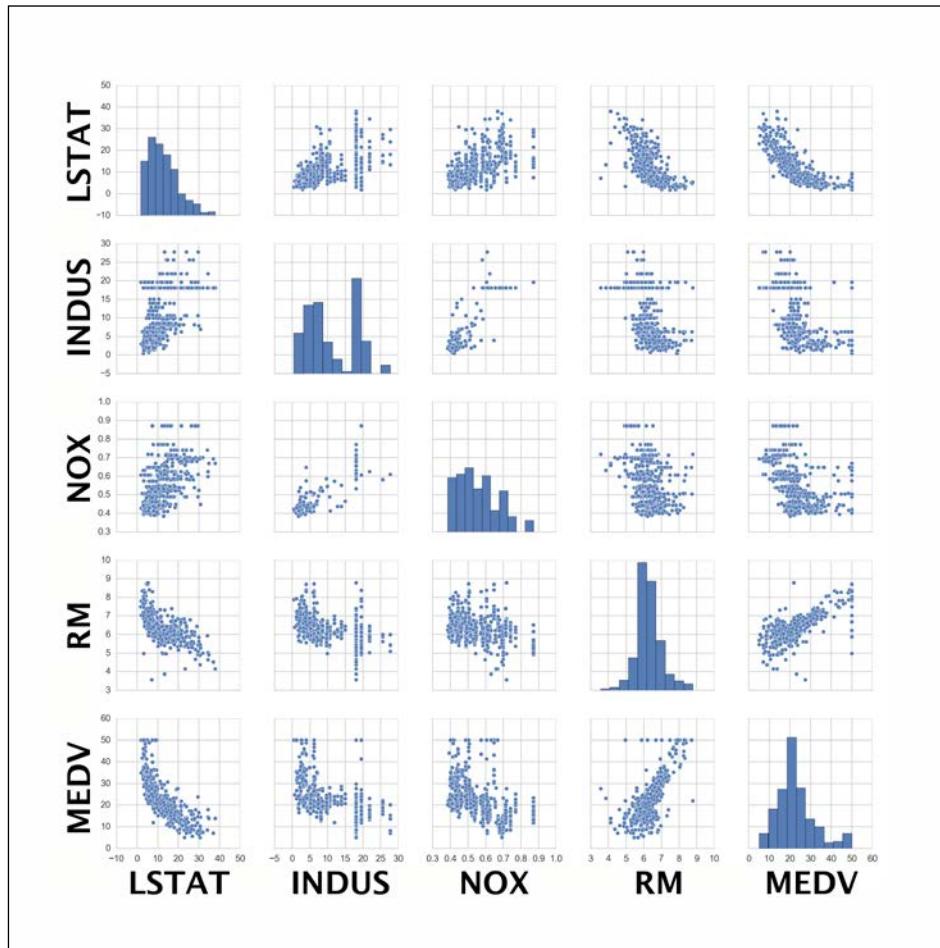
Exploratory Data Analysis (EDA) is an important and recommended first step prior to the training of a machine learning model. In the rest of this section, we will use some simple yet useful techniques from the graphical EDA toolbox that may help us to visually detect the presence of outliers, the distribution of the data, and the relationships between features.

First, we will create a *scatterplot matrix* that allows us to visualize the pair-wise correlations between the different features in this dataset in one place. To plot the scatterplot matrix, we will use the `pairplot` function from the `seaborn` library (`http://stanford.edu/~mwaskom/software/seaborn/`), which is a Python library for drawing statistical plots based on matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> sns.set(style='whitegrid', context='notebook')
```

```
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
>>> sns.pairplot(df[cols], size=2.5)
>>> plt.show()
```

As we can see in the following figure, the scatterplot matrix provides us with a useful graphical summary of the relationships in a dataset:

> Importing the seaborn library modifies the default aesthetics of matplotlib for the current Python session. If you do not want to use seaborn's style settings, you can reset the matplotlib settings by executing the following command:
>
> ```
> >>> sns.reset_orig()
> ```

Due to space constraints and for purposes of readability, we only plotted five columns from the dataset: **LSTAT**, **INDUS**, **NOX**, **RM**, and **MEDV**. However, you are encouraged to create a scatterplot matrix of the whole `DataFrame` to further explore the data.

Using this scatterplot matrix, we can now quickly eyeball how the data is distributed and whether it contains outliers. For example, we can see that there is a linear relationship between **RM** and the housing prices **MEDV** (the fifth column of the fourth row). Furthermore, we can see in the histogram (the lower right subplot in the scatter plot matrix) that the **MEDV** variable seems to be normally distributed but contains several outliers.

> Note that in contrast to common belief, training a linear regression model does not require that the explanatory or target variables are normally distributed. The normality assumption is only a requirement for certain statistical tests and hypothesis tests that are beyond the scope of this book (Montgomery, D. C., Peck, E. A., and Vining, G. G. *Introduction to linear regression analysis*. John Wiley and Sons, 2012, pp.318–319).

To quantify the linear relationship between the features, we will now create a correlation matrix. A correlation matrix is closely related to the covariance matrix that we have seen in the section about **principal component analysis** (**PCA**) in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*. Intuitively, we can interpret the correlation matrix as a rescaled version of the covariance matrix. In fact, the correlation matrix is identical to a covariance matrix computed from standardized data.

The correlation matrix is a square matrix that contains the **Pearson product-moment correlation coefficients** (often abbreviated as **Pearson's r**), which measure the linear dependence between pairs of features. The correlation coefficients are bounded to the range -1 and 1. Two features have a perfect positive correlation if $r = 1$, no correlation if $r = 0$, and a perfect negative correlation if $r = -1$, respectively. As mentioned previously, Pearson's correlation coefficient can simply be calculated as the covariance between two features x and y (numerator) divided by the product of their standard deviations (denominator):

$$r = \frac{\sum_{i=1}^{n}\left[\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)\right]}{\sqrt{\sum_{i=1}^{n}\left(x^{(i)} - \mu_x\right)^2}\sqrt{\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Here, μ denotes the sample mean of the corresponding feature, σ_{xy} is the covariance between the features x and y, and σ_x and σ_y are the features' standard deviations, respectively.

We can show that the covariance between standardized features is in fact equal to their linear correlation coefficient.

Let's first standardize the features x and y, to obtain their z-scores which we will denote as x' and y', respectively:

$$x' = \frac{x - \mu_x}{\sigma_x}, y' = \frac{y - \mu_y}{\sigma_y}$$

Remember that we calculate the (population) covariance between two features as follows:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Since standardization centers a feature variable at mean 0, we can now calculate the covariance between the scaled features as follows:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x'-0\right)\left(y'-0\right)$$

Through resubstitution, we get the following result:

$$\frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right)$$

$$\frac{1}{n \cdot \sigma_x \sigma_y}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$
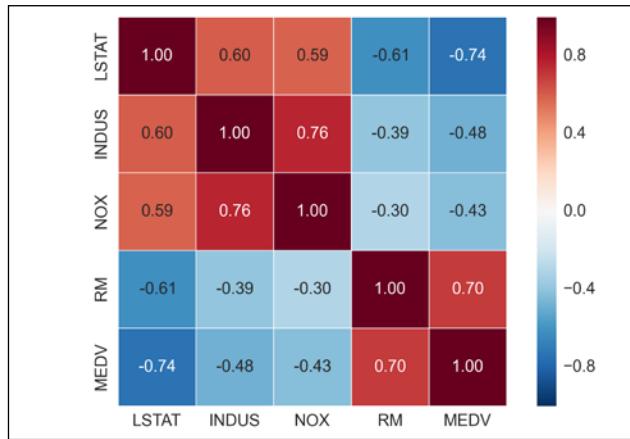
We can simplify it as follows:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

In the following code example, we will use NumPy's `corrcoef` function on the five feature columns that we previously visualized in the scatterplot matrix, and we will use seaborn's `heatmap` function to plot the correlation matrix array as a heat map:

```
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> sns.set(font_scale=1.5)
>>> hm = sns.heatmap(cm,
...                 cbar=True,
...                 annot=True,
...                 square=True,
...                 fmt='.2f',
...                 annot_kws={'size': 15},
...                 yticklabels=cols,
...                 xticklabels=cols)
>>> plt.show()
```

As we can see in the resulting figure, the correlation matrix provides us with another useful summary graphic that can help us to select features based on their respective linear correlations:

To fit a linear regression model, we are interested in those features that have a high correlation with our target variable **MEDV**. Looking at the preceding correlation matrix, we see that our target variable **MEDV** shows the largest correlation with the **LSTAT** variable (-0.74). However, as you might remember from the scatterplot matrix, there is a clear nonlinear relationship between **LSTAT** and **MEDV**. On the other hand, the correlation between **RM** and **MEDV** is also relatively high (0.70) and given the linear relationship between those two variables that we observed in the scatterplot, **RM** seems to be a good choice for an explanatory variable to introduce the concepts of a simple linear regression model in the following section.

Implementing an ordinary least squares linear regression model

At the beginning of this chapter, we discussed that linear regression can be understood as finding the best-fitting straight line through the sample points of our training data. However, we have neither defined the term *best-fitting* nor have we discussed the different techniques of fitting such a model. In the following subsections, we will fill in the missing pieces of this puzzle using the **Ordinary Least Squares** (**OLS**) method to estimate the parameters of the regression line that minimizes the sum of the squared vertical distances (residuals or errors) to the sample points.

Solving regression for regression parameters with gradient descent

Consider our implementation of the **ADAptive LInear NEuron** (**Adaline**) from *Chapter 2*, *Training Machine Learning Algorithms for Classification*; we remember that the artificial neuron uses a linear activation function and we defined a cost function $J(\cdot)$, which we minimized to learn the weights via optimization algorithms, such as **Gradient Descent** (**GD**) and **Stochastic Gradient Descent** (**SGD**). This cost function in Adaline is the **Sum of Squared Errors** (**SSE**). This is identical to the OLS cost function that we defined:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Here, $\hat{y}$ is the predicted value $\hat{y} = w^T x$ (note that the term $1/2$ is just used for convenience to derive the update rule of GD). Essentially, OLS linear regression can be understood as Adaline without the unit step function so that we obtain continuous target values instead of the class labels -1 and 1. To demonstrate the similarity, let's take the GD implementation of *Adaline* from *Chapter 2*, *Training Machine Learning Algorithms for Classification*, and remove the unit step function to implement our first linear regression model:

```
class LinearRegressionGD(object):

    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
```

```
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return self.net_input(X)
```

If you need a refresher about how the weights are being updated—taking a step in the opposite direction of the gradient—please revisit the Adaline section in *Chapter 2, Training Machine Learning Algorithms for Classification*.
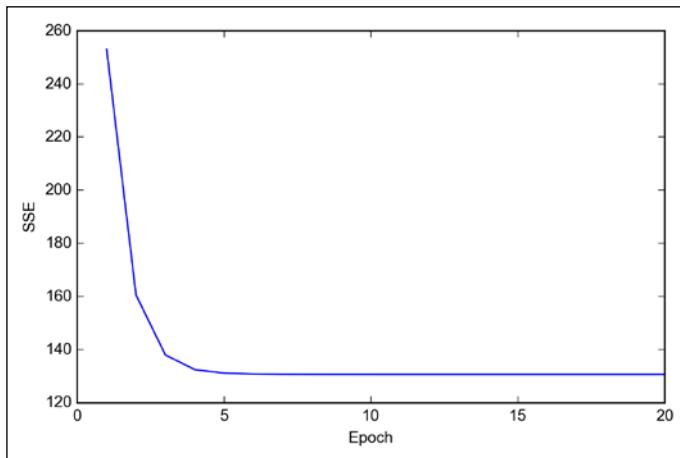
To see our `LinearRegressionGD` regressor in action, let's use the RM (number of rooms) variable from the Housing Data Set as the explanatory variable to train a model that can predict MEDV (the housing prices). Furthermore, we will standardize the variables for better convergence of the GD algorithm. The code is as follows:

```
>>> X = df[['RM']].values
>>> y = df['MEDV'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y)
>>> lr = LinearRegressionGD()
>>> lr.fit(X_std, y_std)
```

We discussed in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, that it is always a good idea to plot the cost as a function of the number of epochs (passes over the training dataset) when we are using optimization algorithms, such as gradient descent, to check for convergence. To cut a long story short, let's plot the cost against the number of epochs to check if the linear regression has converged:

```
>>> plt.plot(range(1, lr.n_iter+1), lr.cost_)
>>> plt.ylabel('SSE')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

As we can see in the following plot, the GD algorithm converged after the fifth epoch:
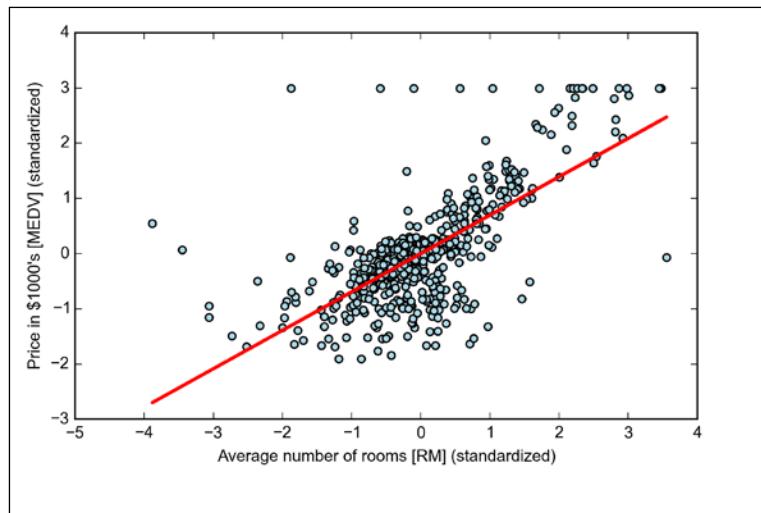
Next, let's visualize how well the linear regression line fits the training data. To do so, we will define a simple helper function that will plot a scatterplot of the training samples and add the regression line:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='blue')
...     plt.plot(X, model.predict(X), color='red')
...     return None
```

Now, we will use this `lin_regplot` function to plot the number of rooms against house prices:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel('Average number of rooms [RM] (standardized)')
>>> plt.ylabel('Price in $1000\'s [MEDV] (standardized)')
>>> plt.show()
```

As we can see in the following plot, the linear regression line reflects the general trend that house prices tend to increase with the number of rooms:

Although this observation makes intuitive sense, the data also tells us that the number of rooms does not explain the house prices very well in many cases. Later in this chapter, we will discuss how to quantify the performance of a regression model. Interestingly, we also observe a curious line $y = 3$, which suggests that the prices may have been clipped. In certain applications, it may also be important to report the predicted outcome variables on their original scale. To scale the predicted price outcome back on the **Price in $1000's** axes, we can simply apply the `inverse_transform` method of the `StandardScaler`:

```
>>> num_rooms_std = sc_x.transform([5.0])
>>> price_std = lr.predict(num_rooms_std)
>>> print("Price in $1000's: %.3f" % \
...        sc_y.inverse_transform(price_std))
Price in $1000's: 10.840
```

In the preceding code example, we used the previously trained linear regression model to predict the price of a house with five rooms. According to our model, such a house is worth $10,840.

On a side note, it is also worth mentioning that we technically don't have to update the weights of the intercept if we are working with standardized variables since the *y* axis intercept is always 0 in those cases. We can quickly confirm this by printing the weights:

```
>>> print('Slope: %.3f' % lr.w_[1])
Slope: 0.695
>>> print('Intercept: %.3f' % lr.w_[0])
Intercept: -0.000
```

Estimating the coefficient of a regression model via scikit-learn
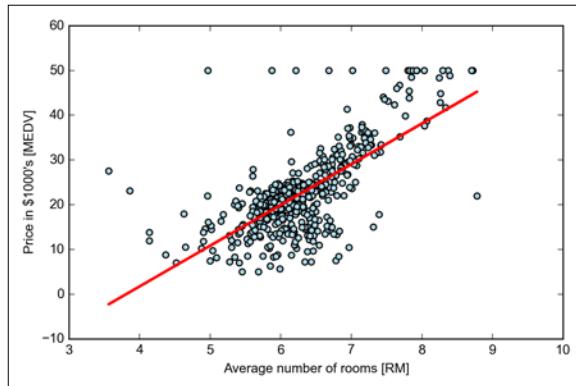
In the previous section, we implemented a working model for regression analysis. However, in a real-world application, we may be interested in more efficient implementations, for example, scikit-learn's `LinearRegression` object that makes use of the **LIBLINEAR** library and advanced optimization algorithms that work better with unstandardized variables. This is sometimes desirable for certain applications:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> print('Slope: %.3f' % slr.coef_[0])
Slope: 9.102
>>> print('Intercept: %.3f' % slr.intercept_)
Intercept: -34.671
```

As we can see by executing the preceding code, scikit-learn's `LinearRegression` model fitted with the unstandardized **RM** and **MEDV** variables yielded different model coefficients. Let's compare it to our own GD implementation by plotting MEDV against RM:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.show()w
```

Now, when we plot the training data and our fitted model by executing the code above, we can see that the overall result looks identical to our GD implementation:

As an alternative to using machine learning libraries, there is also a closed-form solution for solving OLS involving a system of linear equations that can be found in most introductory statistics textbooks:

$$w = \left(X^T X \right)^{-1} X^T y$$

We can implement it in Python as follows:

```
# adding a column vector of "ones"
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print('Slope: %.3f' % w[1])
Slope: 9.102
>>> print('Intercept: %.3f' % w[0])
Intercept: -34.671
```

The advantage of this method is that it is guaranteed to find the optimal solution analytically. However, if we are working with very large datasets, it can be computationally too expensive to invert the matrix in this formula (sometimes also called the **normal equation**) or the sample matrix may be singular (non-invertible), which is why we may prefer iterative methods in certain cases.

If you are interested in more information on how to obtain the normal equations, I recommend you take a look at Dr. Stephen Pollock's chapter, *The Classical Linear Regression Model* from his lectures at the University of Leicester, which are available for free at `http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf`.

Fitting a robust regression model using RANSAC

Linear regression models can be heavily impacted by the presence of outliers. In certain situations, a very small subset of our data can have a big effect on the estimated model coefficients. There are many statistical tests that can be used to detect outliers, which are beyond the scope of the book. However, removing outliers always requires our own judgment as a data scientist, as well as our domain knowledge.

As an alternative to throwing out outliers, we will look at a robust method of regression using the **RANdom SAmple Consensus** (**RANSAC**) algorithm, which fits a regression model to a subset of the data, the so-called *inliers*.

We can summarize the iterative RANSAC algorithm as follows:

1. Select a random number of samples to be inliers and fit the model.
2. Test all other data points against the fitted model and add those points that fall within a user-given tolerance to the inliers.
3. Refit the model using all inliers.
4. Estimate the error of the fitted model versus the inliers.
5. Terminate the algorithm if the performance meets a certain user-defined threshold or if a fixed number of iterations has been reached; go back to step 1 otherwise.

Let's now wrap our linear model in the RANSAC algorithm using scikit-learn's `RANSACRegressor` object:
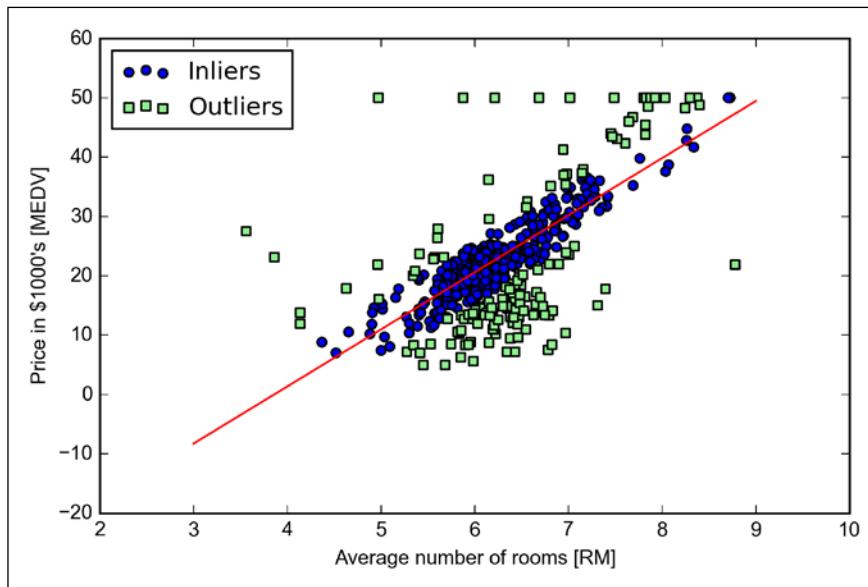
```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(LinearRegression(),
...             max_trials=100,
...             min_samples=50,
...             residual_metric=lambda x: np.sum(np.abs(x), axis=1),
...             residual_threshold=5.0,
...             random_state=0)
>>> ransac.fit(X, y)
```

We set the maximum number of iterations of the `RANSACRegressor` to 100, and using `min_samples=50`, we set the minimum number of the randomly chosen samples to be at least 50. Using the `residual_metric` parameter, we provided a callable `lambda` function that simply calculates the absolute vertical distances between the fitted line and the sample points. By setting the `residual_threshold` parameter to `5.0`, we only allowed samples to be included in the inlier set if their vertical distance to the fitted line is within 5 distance units, which works well on this particular dataset. By default, scikit-learn uses the MAD estimate to select the inlier threshold, where **MAD** stands for the **Median Absolute Deviation** of the target values y. However, the choice of an appropriate value for the inlier threshold is problem-specific, which is one disadvantage of RANSAC. Many different approaches have been developed over the recent years to select a good inlier threshold automatically. You can find a detailed discussion in R. Toldo and A. Fusiello's. *Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting* (in Image Analysis and Processing–ICIAP 2009, pages 123–131. Springer, 2009).

After we have fitted the RANSAC model, let's obtain the inliers and outliers from the fitted RANSAC linear regression model and plot them together with the linear fit:

```
>>> inlier_mask = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...             c='blue', marker='o', label='Inliers')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...             c='lightgreen', marker='s', label='Outliers')
>>> plt.plot(line_X, line_y_ransac, color='red')
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

As we can see in the following scatterplot, the linear regression model was fitted on the detected set of inliers shown as circles:

When we print the slope and intercept of the model executing the following code, we can see that the linear regression line is slightly different from the fit that we obtained in the previous section without RANSAC:

```
>>> print('Slope: %.3f' % ransac.estimator_.coef_[0])
Slope: 9.621
>>> print('Intercept: %.3f' % ransac.estimator_.intercept_)
Intercept: -37.137
```

Using RANSAC, we reduced the potential effect of the outliers in this dataset, but we don't know if this approach has a positive effect on the predictive performance for unseen data. Thus, in the next section we will discuss how to evaluate a regression model for different approaches, which is a crucial part of building systems for predictive modeling.

Evaluating the performance of linear regression models

In the previous section, we discussed how to fit a regression model on training data. However, you learned in previous chapters that it is crucial to test the model on data that it hasn't seen during training to obtain an unbiased estimate of its performance.

As we remember from *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we want to split our dataset into separate training and test datasets where we use the former to fit the model and the latter to evaluate its performance to generalize to unseen data. Instead of proceeding with the simple regression model, we will now use all variables in the dataset and train a multiple regression model:
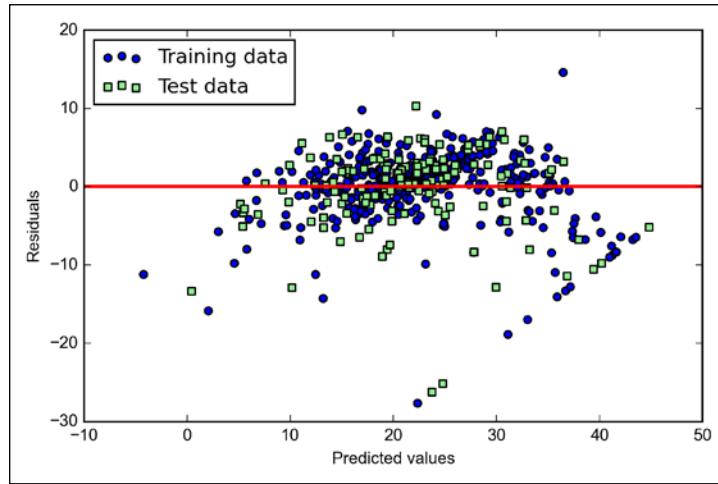
```
>>> from sklearn.cross_validation import train_test_split
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...          X, y, test_size=0.3, random_state=0)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

Since our model uses multiple explanatory variables, we can't visualize the linear regression line (or hyperplane to be precise) in a two-dimensional plot, but we can plot the residuals (the differences or vertical distances between the actual and predicted values) versus the predicted values to diagnose our regression model. Those **residual plots** are a commonly used graphical analysis for diagnosing regression models to detect nonlinearity and outliers, and to check if the errors are randomly distributed.

Using the following code, we will now plot a residual plot where we simply subtract the true target variables from our predicted responses:

```
>>> plt.scatter(y_train_pred, y_train_pred - y_train,
...             c='blue', marker='o', label='Training data')
>>> plt.scatter(y_test_pred,  y_test_pred - y_test,
...             c='lightgreen', marker='s', label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

After executing the code, we should see a residual plot with a line passing through the *x* axis origin as shown here:

In the case of a perfect prediction, the residuals would be exactly zero, which we will probably never encounter in realistic and practical applications. However, for a good regression model, we would expect that the errors are randomly distributed and the residuals should be randomly scattered around the centerline. If we see patterns in a residual plot, it means that our model is unable to capture some explanatory information, which is leaked into the residuals as we can slightly see in our preceding residual plot. Furthermore, we can also use residual plots to detect outliers, which are represented by the points with a large deviation from the centerline.

Another useful quantitative measure of a model's performance is the so-called **Mean Squared Error** (**MSE**), which is simply the average value of the SSE cost function that we minimize to fit the linear regression model. The MSE is useful to for comparing different regression models or for tuning their parameters via a grid search and cross-validation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Execute the following code:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
```

We will see that the MSE on the training set is 19.96, and the MSE of the test set is much larger with a value of 27.20, which is an indicator that our model is overfitting the training data.

Sometimes it may be more useful to report the coefficient of determination (R^2), which can be understood as a standardized version of the MSE, for better interpretability of the model performance. In other words, R^2 is the fraction of response variance that is captured by the model. The R^2 value is defined as follows:

$$R^2 = 1 - \frac{SSE}{SST}$$

Here, SSE is the sum of squared errors and SST is the total sum of squares $SST = \sum_{i=1}^{n} \left(y^{(i)} - \mu_y \right)^2$, or in other words, it is simply the variance of the response.

Let's quickly show that R^2 is indeed just a rescaled version of the MSE:

$$R^2 = 1 - \frac{SSE}{SST}$$

$$1 - \frac{\frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)} \right)^2}{\frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - \mu_y \right)^2}$$

$$1 - \frac{MSE}{Var(y)}$$

For the training dataset, R^2 is bounded between 0 and 1, but it can become negative for the test set. If $R^2 = 1$, the model fits the data perfectly with a corresponding $MSE = 0$.

Evaluated on the training data, the R^2 of our model is 0.765, which doesn't sound too bad. However, the R^2 on the test dataset is only 0.673, which we can compute by executing the following code:

```
>>> from sklearn.metrics import r2_score
>>> print('R^2 train: %.3f, test: %.3f' %
```

```
...           (r2_score(y_train, y_train_pred),
...            r2_score(y_test, y_test_pred)))
```

Using regularized methods for regression

As we discussed in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, regularization is one approach to tackle the problem of overfitting by adding additional information, and thereby shrinking the parameter values of the model to induce a penalty against complexity. The most popular approaches to regularized linear regression are the so-called **Ridge Regression**, **Least Absolute Shrinkage and Selection Operator** (**LASSO**) and **Elastic Net** method.

Ridge regression is an L2 penalized model where we simply add the squared sum of the weights to our least-squares cost function:

$$J(w)_{Ridge} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \parallel w \parallel_2^2$$

Here:

$$L2: \quad \lambda \parallel w \parallel_2^2 = \lambda \sum_{j=1}^{m} w_j^{\,2}$$

By increasing the value of the hyperparameter λ, we increase the regularization strength and shrink the weights of our model. Please note that we don't regularize the intercept term w_0.

An alternative approach that can lead to sparse models is the LASSO. Depending on the regularization strength, certain weights can become zero, which makes the LASSO also useful as a supervised feature selection technique:

$$J(w)_{LASSO} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \parallel w \parallel_1$$

Here:

$$L1: \lambda \parallel w \parallel_1 = \lambda \sum_{j=1}^{m} \left| w_j \right|$$

However, a limitation of the LASSO is that it selects at most n variables if $m > n$. A compromise between Ridge regression and the LASSO is the Elastic Net, which has a L1 penalty to generate sparsity and a L2 penalty to overcome some of the limitations of the LASSO, such as the number of selected variables.

$$J(w)_{ElasticNet} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1 \sum_{j=1}^{m} w_j^2 + \lambda_2 \sum_{j=1}^{m} \left|w_j\right|$$

Those regularized regression models are all available via scikit-learn, and the usage is similar to the regular regression model except that we have to specify the regularization strength via the parameter λ, for example, optimized via k-fold cross-validation.

A Ridge Regression model can be initialized as follows:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Note that the regularization strength is regulated by the parameter `alpha`, which is similar to the parameter λ. Likewise, we can initialize a LASSO regressor from the `linear_model` submodule:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Lastly, the `ElasticNet` implementation allows us to vary the L1 to L2 ratio:

```
>>> from sklearn.linear_model import ElasticNet
>>> lasso = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

For example, if we set `l1_ratio` to `1.0`, the `ElasticNet` regressor would be equal to LASSO regression. For more detailed information about the different implementations of linear regression, please see the documentation at `http://scikit-learn.org/stable/modules/linear_model.html`.

Turning a linear regression model into a curve – polynomial regression

In the previous sections, we assumed a linear relationship between explanatory and response variables. One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms:

$$y = w_0 + w_1 x + w_2 x^2 x^2 + \ldots + w_d x^d$$

Here, d denotes the degree of the polynomial. Although we can use polynomial regression to model a nonlinear relationship, it is still considered a multiple linear regression model because of the linear regression coefficients w.

We will now discuss how to use the `PolynomialFeatures` transformer class from scikit-learn to add a quadratic term ($d=2$) to a simple regression problem with one explanatory variable, and compare the polynomial to the linear fit. The steps are as follows:

1. Add a second degree polynomial term:

    ```
    from sklearn.preprocessing import PolynomialFeatures
    >>> X = np.array([258.0, 270.0, 294.0,
    …                          320.0, 342.0, 368.0,
    …                          396.0, 446.0, 480.0,
    …                          586.0])[:, np.newaxis]

    >>> y = np.array([236.4, 234.4, 252.8,
    …                          298.6, 314.2, 342.2,
    …                          360.8, 368.0, 391.2,
    …                          390.8])
    >>> lr = LinearRegression()
    >>> pr = LinearRegression()
    >>> quadratic = PolynomialFeatures(degree=2)
    >>> X_quad = quadratic.fit_transform(X)
    ```
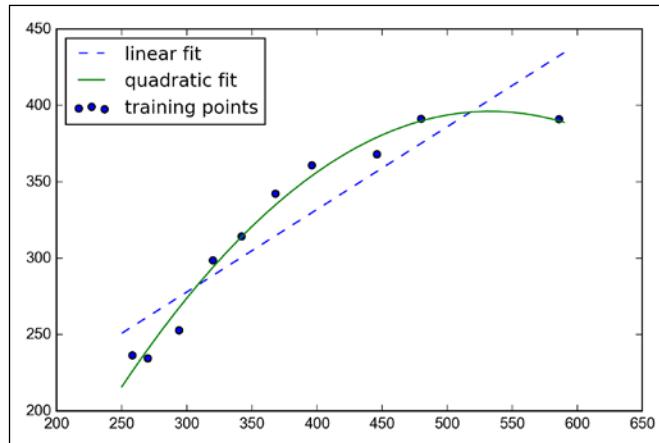
2. Fit a simple linear regression model for comparison:

    ```
    >>> lr.fit(X, y)
    >>> X_fit = np.arange(250,600,10)[:, np.newaxis]
    >>> y_lin_fit = lr.predict(X_fit)
    ```

3. Fit a multiple regression model on the transformed features for polynomial regression:

    ```
    >>> pr.fit(X_quad, y)
    >>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
    Plot the results:
    >>> plt.scatter(X, y, label='training points')
    >>> plt.plot(X_fit, y_lin_fit,
    ...          label='linear fit', linestyle='--')
    >>> plt.plot(X_fit, y_quad_fit,
    ...          label='quadratic fit')
    >>> plt.legend(loc='upper left')
    >>> plt.show()
    ```

In the resulting plot, we can see that the polynomial fit captures the relationship between the response and explanatory variable much better than the linear fit:


```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('Training MSE linear: %.3f, quadratic: %.3f' % (
...         mean_squared_error(y, y_lin_pred),
...         mean_squared_error(y, y_quad_pred)))
Training MSE linear: 569.780, quadratic: 61.330
>>> print('Training R^2 linear: %.3f, quadratic: %.3f' % (
...         r2_score(y, y_lin_pred),
...         r2_score(y, y_quad_pred)))
Training R^2 linear: 0.832, quadratic: 0.982
```

As we can see after executing the preceding code, the MSE decreased from 570 (linear fit) to 61 (quadratic fit), and the coefficient of determination reflects a closer fit to the quadratic model ($R^2 = 0.982$) as opposed to the linear fit ($R^2 = 0.832$) in this particular toy problem.

Modeling nonlinear relationships in the Housing Dataset

After we discussed how to construct polynomial features to fit nonlinear relationships in a toy problem, let's now take a look at a more concrete example and apply those concepts to the data in the *Housing Dataset*. By executing the following code, we will model the relationship between house prices and LSTAT (percent lower status of the population) using second degree (quadratic) and third degree (cubic) polynomials and compare it to a linear fit.

The code is as follows:

```
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> regr = LinearRegression()

# create polynomial features
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)

# linear fit
>>> X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]
>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))

# quadratic fit
>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))

# cubic fit
>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))

# plot results
>>> plt.scatter(X, y,
...             label='training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label='linear (d=1), $R^2=%.2f$'
...             % linear_r2,
...          color='blue',
...          lw=2,
...          linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...          label='quadratic (d=2), $R^2=%.2f$'
...             % quadratic_r2,
...          color='red',
...          lw=2,
...          linestyle='-')
```
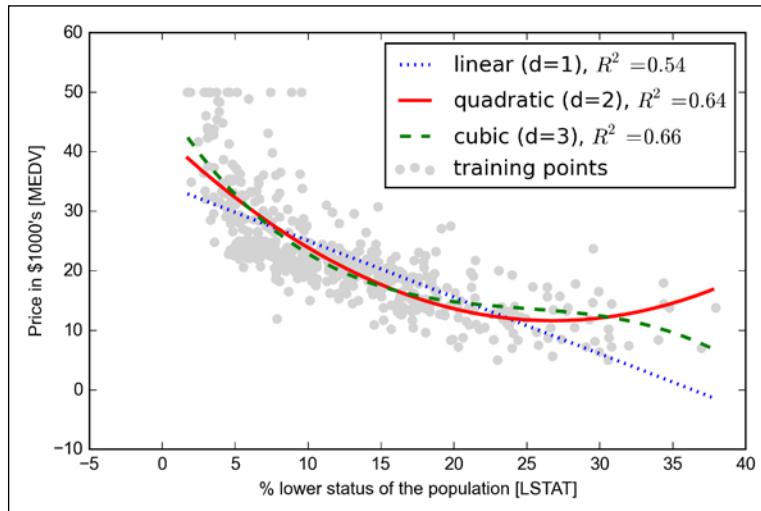
```
>>> plt.plot(X_fit, y_cubic_fit,
...          label='cubic (d=3), $R^2=%.2f$'
...            % cubic_r2,
...          color='green',
...          lw=2,
...          linestyle='--')
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

As we can see in the resulting plot, the cubic fit captures the relationship between the house prices and **LSTAT** better than the linear and quadratic fit. However, we should be aware that adding more and more polynomial features increases the complexity of a model and therefore increases the chance of overfitting. Thus, in practice, it is always recommended that you evaluate the performance of the model on a separate test dataset to estimate the generalization performance:

In addition, polynomial features are not always the best choice for modeling nonlinear relationships. For example, just by looking at the **MEDV-LSTAT** scatterplot, we could propose that a log transformation of the **LSTAT** feature variable and the square root of **MEDV** may project the data onto a linear feature space suitable for a linear regression fit. Let's test this hypothesis by executing the following code:

```
# transform features
>>> X_log = np.log(X)
```
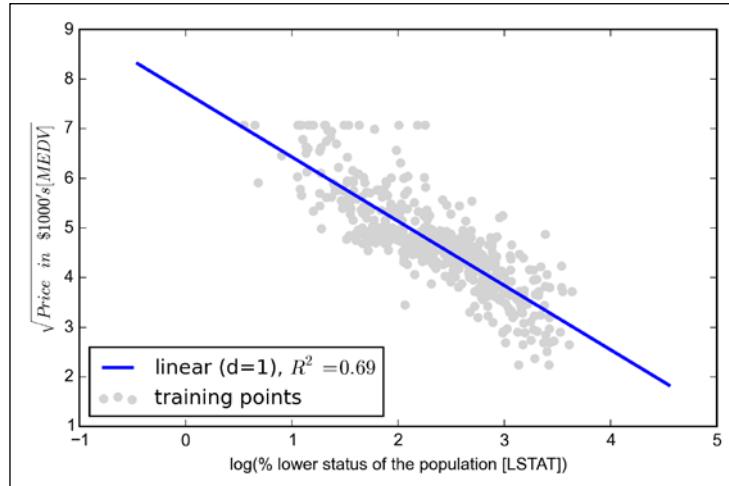
```
>>> y_sqrt = np.sqrt(y)

# fit features
>>> X_fit = np.arange(X_log.min()-1,
...                    X_log.max()+1, 1)[:, np.newaxis]
>>> regr = regr.fit(X_log, y_sqrt)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y_sqrt, regr.predict(X_log))

# plot results
>>> plt.scatter(X_log, y_sqrt,
...             label='training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label='linear (d=1), $R^2=%.2f$' % linear_r2,
...          color='blue',
...          lw=2)
>>> plt.xlabel('log(% lower status of the population [LSTAT])')
>>> plt.ylabel('$\sqrt{Price \; in \; \$1000\'s [MEDV]}$')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

After transforming the explanatory onto the log space and taking the square root of the target variables, we were able to capture the relationship between the two variables with a linear regression line that seems to fit the data better ($R^2 = 0.69$) than any of the polynomial feature transformations previously:

Dealing with nonlinear relationships using random forests

In this section, we are going to take a look at **random forest** regression, which is conceptually different from the previous regression models in this chapter. A random forest, which is an ensemble of multiple **decision trees**, can be understood as the sum of piecewise linear functions in contrast to the global linear and polynomial regression models that we discussed previously. In other words, via the decision tree algorithm, we are subdividing the input space into smaller regions that become more *manageable*.

Decision tree regression

An advantage of the decision tree algorithm is that it does not require any transformation of the features if we are dealing with nonlinear data. We remember from *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, that we grow a decision tree by iteratively splitting its nodes until the leaves are pure or a stopping criterion is satisfied. When we used decision trees for classification, we defined entropy as a measure of impurity to determine which feature split maximizes the **Information Gain** (**IG**), which can be defined as follows for a binary split:

$$IG\left(D_p, x_i\right) = I\left(D_p\right) - \frac{N_{left}}{N_p} I\left(D_{left}\right) - \frac{N_{right}}{N_p} I\left(D_{right}\right)$$

Here, x is the feature to perform the split, N_p is the number of samples in the parent node, I is the impurity function, D_p is the subset of training samples in the parent node, and D_{left} and D_{right} are the subsets of training samples in the left and right child node after the split. Remember that our goal is to find the feature split that maximizes the information gain, or in other words, we want to find the feature split that reduces the impurities in the child nodes. In *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we used *entropy* as a measure of impurity, which is a useful criterion for classification. To use a decision tree for regression, we will replace entropy as the impurity measure of a node t by the MSE:

$$I\left(t\right) = MSE\left(t\right) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t\right)^2$$
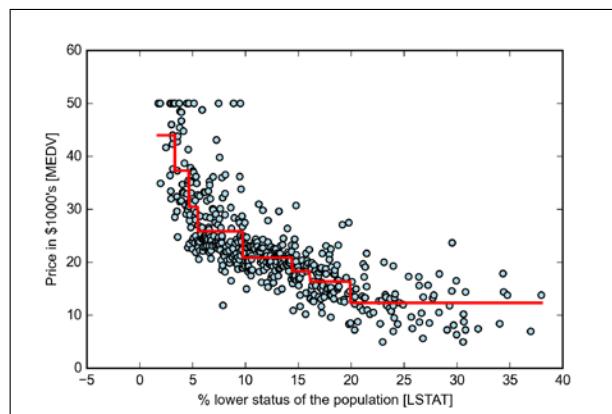
Here, N_t is the number of training samples at node t, D_t is the training subset at node t, $y^{(i)}$ is the true target value, and $\hat{y}_t$ is the predicted target value (sample mean):

$$\hat{y}_t = \frac{1}{N} \sum_{i \in D_t} y^{(i)}$$

In the context of decision tree regression, the MSE is often also referred to as within-node variance, which is why the splitting criterion is also better known as *variance reduction*. To see what the line fit of a decision tree looks like, let's use the `DecisionTreeRegressor` implemented in scikit-learn to model the nonlinear relationship between the **MEDV** and **LSTAT** variables:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
>>> sort_idx = X.flatten().argsort()
   >>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.show()
```

As we can see from the resulting plot, the decision tree captures the general trend in the data. However, a limitation of this model is that it does not capture the continuity and differentiability of the desired prediction. In addition, we need to be careful about choosing an appropriate value for the depth of the tree to not overfit or underfit the data; here, a depth of 3 seems to be a good choice:

In the next section, we will take a look at a more robust way for fitting regression trees: random forests.

Random forest regression

As we discussed in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, the random forest algorithm is an ensemble technique that combines multiple decision trees. A random forest usually has a better generalization performance than an individual decision tree due to randomness that helps to decrease the model variance. Other advantages of random forests are that they are less sensitive to outliers in the dataset and don't require much parameter tuning. The only parameter in random forests that we typically need to experiment with is the number of trees in the ensemble. The basic random forests algorithm for regression is almost identical to the random forest algorithm for classification that we discussed in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. The only difference is that we use the MSE criterion to grow the individual decision trees, and the predicted target variable is calculated as the average prediction over all decision trees.

Now, let's use all the features in the Housing Dataset to fit a random forest regression model on 60 percent of the samples and evaluate its performance on the remaining 40 percent. The code is as follows:

```
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test =\
...       train_test_split(X, y,
...                        test_size=0.4,
...                        random_state=1)

>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(                          ..
.                                  n_estimators=1000,
...                                 criterion='mse',
...                                 random_state=1,
...                                 n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
```
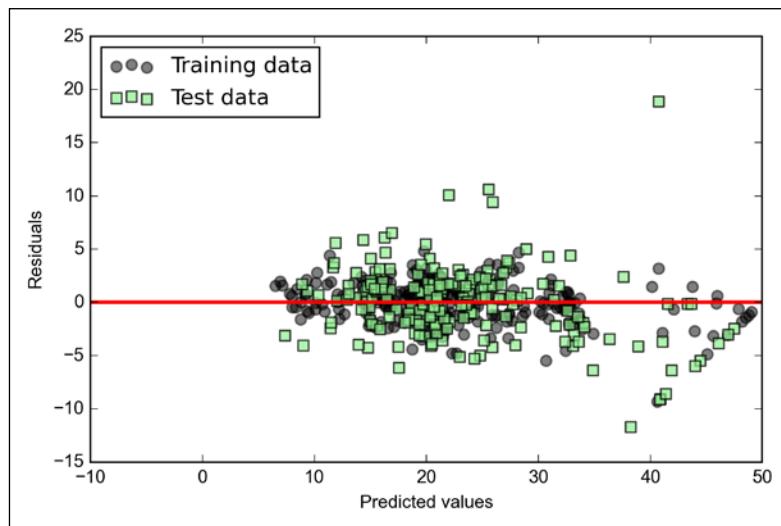
```
>>> print('MSE train: %.3f, test: %.3f' % (
...          mean_squared_error(y_train, y_train_pred),
...          mean_squared_error(y_test, y_test_pred)))
>>> print('R^2 train: %.3f, test: %.3f' % (
...          r2_score(y_train, y_train_pred),
...          r2_score(y_test, y_test_pred)))
MSE train: 1.642, test: 11.635
R^2 train: 0.960, test: 0.871
```

Unfortunately, we see that the random forest tends to overfit the training data. However, it's still able to explain the relationship between the target and explanatory variables relatively well ($R^2 = 0.871$ on the test dataset).

Lastly, let's also take a look at the residuals of the prediction:

```
>>> plt.scatter(y_train_pred,
...              y_train_pred - y_train,
...              c='black',
...              marker='o',
...              s=35,
...              alpha=0.5,
...              label='Training data')
>>> plt.scatter(y_test_pred,
...              y_test_pred - y_test,
...              c='lightgreen',
...              marker='s',
...              s=35,
...              alpha=0.7,
...              label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

As it was already summarized by the R^2 coefficient, we can see that the model fits the training data better than the test data, as indicated by the outliers in the y axis direction. Also, the distribution of the residuals does not seem to be completely random around the zero center point, indicating that the model is not able to capture all the exploratory information. However, the residual plot indicates a large improvement over the residual plot of the linear model that we plotted earlier in this chapter:

In *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we also discussed the kernel trick that can be used in combination with **support vector machine** (**SVM**) for classification, which is useful if we are dealing with nonlinear problems. Although a discussion is beyond of the scope of this book, SVMs can also be used in nonlinear regression tasks. The interested reader can find more information about Support Vector Machines for regression in an excellent report by S. R. Gunn: S. R. Gunn et al. *Support Vector Machines for Classification and Regression*. (ISIS technical report, 14, 1998). An SVM regressor is also implemented in scikit-learn, and more information about its usage can be found at `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR`.

Summary

At the beginning of this chapter, you learned about using simple linear regression analysis to model the relationship between a single explanatory variable and a continuous response variable. We then discussed a useful explanatory data analysis technique to look at patterns and anomalies in data, which is an important first step in predictive modeling tasks.

We built our first model by implementing linear regression using a gradient-based optimization approach. We then saw how to utilize scikit-learn's linear models for regression and also implement a robust regression technique (RANSAC) as an approach for dealing with outliers. To assess the predictive performance of regression models, we computed the mean sum of squared errors and the related R^2 metric. Furthermore, we also discussed a useful graphical approach to diagnose the problems of regression models: the residual plot.

After we discussed how regularization can be applied to regression models to reduce the model complexity and avoid overfitting, we also introduced several approaches to model nonlinear relationships, including polynomial feature transformation and random forest regressors.

We discussed supervised learning, classification, and regression analysis, in great detail throughout the previous chapters. In the next chapter, we are going to discuss another interesting subfield of machine learning: unsupervised learning. In the next chapter, you will learn how to use cluster analysis for finding hidden structures in data in the absence of target variables.

11

Working with Unlabeled Data – Clustering Analysis

In the previous chapters, we used supervised learning techniques to build machine learning models using data where the answer was already known—the class labels were already available in our training data. In this chapter, we will switch gears and explore cluster analysis, a category of **unsupervised learning** techniques that allows us to discover hidden structures in data where we do not know the right answer upfront. The goal of clustering is to find a natural grouping in data such that items in the same cluster are more similar to each other than those from different clusters.

Given its exploratory nature, clustering is an exciting topic and, in this chapter, you will learn about the following concepts that can help you to organize data into meaningful structures:

- Finding centers of similarity using the popular k-means algorithm
- Using a bottom-up approach to build hierarchical cluster trees
- Identifying arbitrary shapes of objects using a density-based clustering approach

Grouping objects by similarity using k-means

In this section, we will discuss one of the most popular **clustering** algorithms, **k-means**, which is widely used in academia as well as in industry. Clustering (or cluster analysis) is a technique that allows us to find groups of similar objects, objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines.

As we will see in a moment, the k-means algorithm is extremely easy to implement but is also computationally very efficient compared to other clustering algorithms, which might explain its popularity. The k-means algorithm belongs to the category of prototype-based clustering. We will discuss two other categories of clustering, **hierarchical** and **density-based** clustering, later in this chapter. **Prototype-based** clustering means that each cluster is represented by a prototype, which can either be the **centroid** (*average*) of similar points with continuous features, or the **medoid** (the most *representative* or most frequently occurring point) in the case of categorical features. While k-means is very good at identifying clusters of spherical shape, one of the drawbacks of this clustering algorithm is that we have to specify the number of clusters k a priori. An inappropriate choice for k can result in poor clustering performance. Later in this chapter, we will discuss the **elbow** method and **silhouette plots**, which are useful techniques to evaluate the quality of a clustering to help us determine the optimal number of clusters k.

Although k-means clustering can be applied to data in higher dimensions, we will walk through the following examples using a simple two-dimensional dataset for the purpose of visualization:
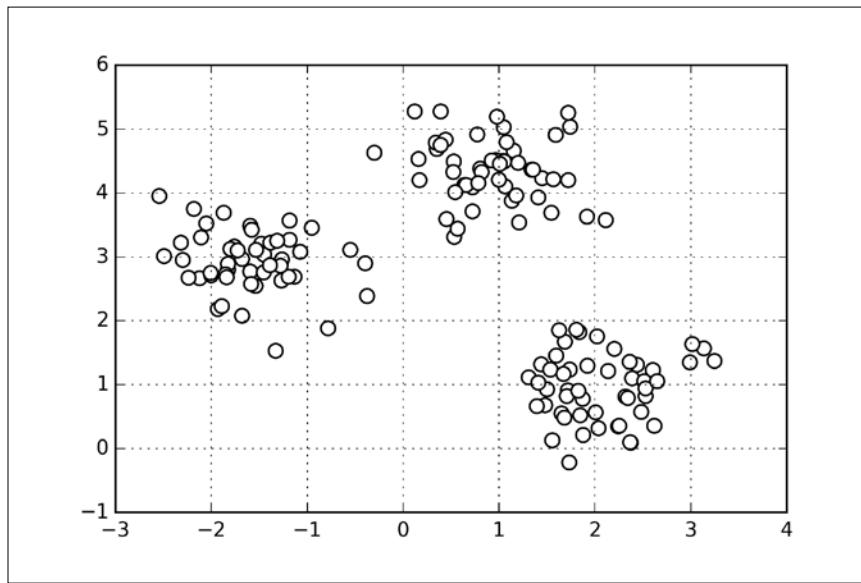
```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=150,
...                   n_features=2,
...                   centers=3,
...                   cluster_std=0.5,
...                   shuffle=True,
...                   random_state=0)

>>> import matplotlib.pyplot as plt
>>> plt.scatter(X[:,0],
...             X[:,1],
...             c='white',
```

```
...                 marker='o',
...                 s=50)
>>> plt.grid()
>>> plt.show()
```

The dataset that we just created consists of 150 randomly generated points that are roughly grouped into three regions with higher density, which is visualized via a two-dimensional scatterplot:

In real-world applications of clustering, we do not have any ground truth category information about those samples; otherwise, it would fall into the category of supervised learning. Thus, our goal is to group the samples based on their feature similarities, which we can be achieved using the k-means algorithm that can be summarized by the following four steps:

1. Randomly pick *k* centroids from the sample points as initial cluster centers.

2. Assign each sample to the nearest centroid $\mu^{(j)}$, $j \in \{1,\dots,k\}$.

3. Move the centroids to the center of the samples that were assigned to it.

4. Repeat steps 2 and 3 until the cluster assignments do not change or a user-defined tolerance or a maximum number of iterations is reached.

Now the next question is *how do we measure similarity between objects?* We can define similarity as the opposite of distance, and a commonly used distance for clustering samples with continuous features is the **squared Euclidean distance** between two points *x* and *y* in *m-dimensional* space:

$$d\left(\boldsymbol{x}, \boldsymbol{y}\right)^2 = \sum_{j=1}^{m}\left(x_j - y_j\right)^2 = \left\|\boldsymbol{x} - \boldsymbol{y}\right\|_2^2$$

Note that, in the preceding equation, the index *j* refers to the *jth* dimension (feature column) of the sample points *x* and *y*. In the rest of this section, we will use the superscripts *i* and *j* to refer to the sample index and cluster index, respectively.

Based on this Euclidean distance metric, we can describe the k-means algorithm as a simple optimization problem, an iterative approach for minimizing the **within-cluster sum of squared errors** (**SSE**), which is sometimes also called **cluster inertia**:

$$SSE = \sum_{i=1}^{n}\sum_{j=1}^{k} w^{(i,j)}\left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)}\right\|_2^2$$

Here, $\boldsymbol{\mu}^{(j)}$ is the representative point (centroid) for cluster *j*, and $w^{(i,j)} = 1$ if the sample $\boldsymbol{x}^{(i)}$ is in cluster *j*; $w^{(i,j)} = 0$ otherwise.

Now that you have learned how the simple k-means algorithm works, let's apply it to our sample dataset using the `KMeans` class from scikit-learn's `cluster` module:

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...             init='random',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
```

Using the preceding code, we set the number of desired clusters to 3; specifying the number of clusters a priori is one of the limitations of k-means. We set `n_init=10` to run the k-means clustering algorithms 10 times independently with different random centroids to choose the final model as the one with the lowest SSE. Via the `max_iter` parameter, we specify the maximum number of iterations for each single run (here, `300`). Note that the k-means implementation in scikit-learn stops early if it converges before the maximum number of iterations is reached.

However, it is possible that k-means does not reach convergence for a particular run, which can be problematic (computationally expensive) if we choose relatively large values for `max_iter`. One way to deal with convergence problems is to choose larger values for `tol`, which is a parameter that controls the tolerance with regard to the changes in the within-cluster sum-squared-error to declare convergence. In the preceding code, we chose a tolerance of `1e-04` (=0.0001).

K-means++

So far, we discussed the classic k-means algorithm that uses a random seed to place the initial centroids, which can sometimes result in bad clusterings or slow convergence if the initial centroids are chosen poorly. One way to address this issue is to run the k-means algorithm multiple times on a dataset and choose the best performing model in terms of the SSE. Another strategy is to place the initial centroids far away from each other via the **k-means++** algorithm, which leads to better and more consistent results than the classic k-means (D. Arthur and S. Vassilvitskii. k-means++: *The Advantages of Careful Seeding*. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007).

The initialization in k-means++ can be summarized as follows:

1. Initialize an empty set $\mathbf{M}$ to store the k centroids being selected.
2. Randomly choose the first centroid $\boldsymbol{\mu}^{(j)}$ from the input samples and assign it to $\mathbf{M}$.
3. For each sample $x^{(i)}$ that is not in $\mathbf{M}$, find the minimum squared distance $d\left(x^{(i)},\mathbf{M}\right)^2$ to any of the centroids in $\mathbf{M}$.
4. To randomly select the next centroid $\boldsymbol{\mu}^{(p)}$, use a weighted probability distribution equal to $\dfrac{d\left(\boldsymbol{\mu}^{(p)},\mathbf{M}\right)^2}{\sum_i d\left(x^{(i)},\mathbf{M}\right)^2}$.
5. Repeat steps 2 and 3 until k centroids are chosen.
6. Proceed with the classic k-means algorithm.

> To use k-means++ with scikit-learn's `KMeans` object, we just need to set the `init` parameter to `k-means++` (the default setting) instead of `random`.

Another problem with k-means is that one or more clusters can be empty. Note that this problem does not exist for k-medoids or fuzzy C-means, an algorithm that we will discuss in the next subsection. However, this problem is accounted for in the current k-means implementation in scikit-learn. If a cluster is empty, the algorithm will search for the sample that is farthest away from the centroid of the empty cluster. Then it will reassign the centroid to be this farthest point.
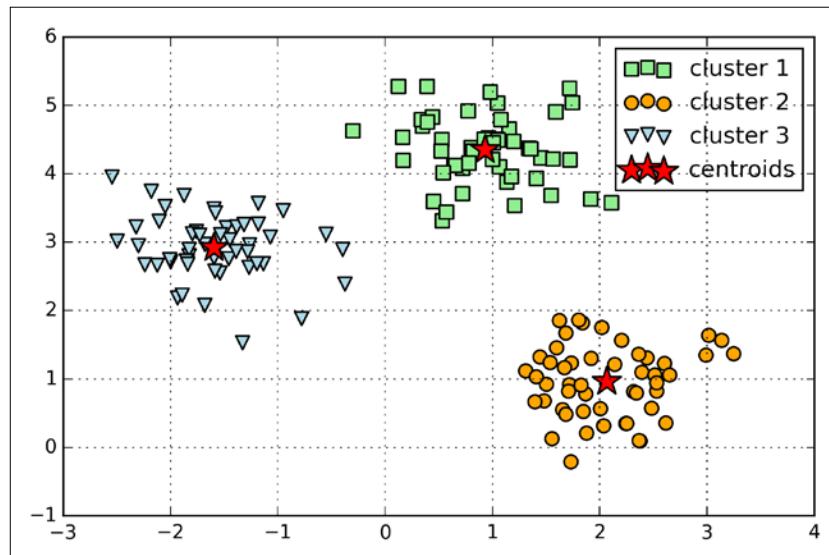
> When we are applying k-means to real-world data using a Euclidean distance metric, we want to make sure that the features are measured on the same scale and apply z-score standardization or min-max scaling if necessary.

After we predicted the cluster labels `y_km` and discussed the challenges of the k-means algorithm, let's now visualize the clusters that k-means identified in the dataset together with the cluster centroids. These are stored under the `centers_` attribute of the fitted `KMeans` object:

```
>>> plt.scatter(X[y_km==0,0],
...             X[y_km==0,1],
...             s=50,
...             c='lightgreen',
...             marker='s',
...             label='cluster 1')
>>> plt.scatter(X[y_km==1,0],
...             X[y_km==1,1],
...             s=50,
...             c='orange',
...             marker='o',
...             label='cluster 2')
>>> plt.scatter(X[y_km==2,0],
...             X[y_km==2,1],
...             s=50,
...             c='lightblue',
...             marker='v',
...             label='cluster 3')
>>> plt.scatter(km.cluster_centers_[:,0],
...             km.cluster_centers_[:,1],
...             s=250,
...             marker='*',
...             c='red',
...             label='centroids')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()
```

In the following scatterplot, we can see that k-means placed the three centroids at the center of each sphere, which looks like a reasonable grouping given this dataset:

Although k-means worked well on this toy dataset, we need to note some of the main challenges of k-means. One of the drawbacks of k-means is that we have to specify the number of clusters *k* a priori, which may not always be so obvious in real-world applications, especially if we are working with a higher dimensional dataset that cannot be visualized. The other properties of k-means are that clusters do not overlap and are not hierarchical, and we also assume that there is at least one item in each cluster.

Hard versus soft clustering

Hard clustering describes a family of algorithms where each sample in a dataset is assigned to exactly one cluster, as in the k-means algorithm that we discussed in the previous subsection. In contrast, algorithms for **soft clustering** (sometimes also called **fuzzy clustering**) assign a sample to one or more clusters. A popular example of soft clustering is the **fuzzy C-means** (**FCM**) algorithm (also called **soft k-means** or **fuzzy k-means**). The original idea goes back to the 1970s where Joseph C. Dunn first proposed an early version of fuzzy clustering to improve k-means (J. C. Dunn. *A Fuzzy Relative of the Isodata Process and its Use in Detecting Compact Well-separated Clusters*. 1973). Almost a decade later, James C. Bedzek published his work on the improvements of the fuzzy clustering algorithm, which is now known as the FCM algorithm (J. C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Springer Science & Business Media, 2013).

The FCM procedure is very similar to k-means. However, we replace the hard cluster assignment by probabilities for each point belonging to each cluster. In k-means, we could express the cluster membership of a sample *x* by a sparse vector of binary values:

$$\begin{bmatrix} \boldsymbol{\mu}^{(1)} \to 0 \\ \boldsymbol{\mu}^{(2)} \to 1 \\ \boldsymbol{\mu}^{(3)} \to 0 \end{bmatrix}$$

Here, the index position with value 1 indicates the cluster centroid $\boldsymbol{\mu}^{(j)}$ the sample is assigned to (assuming $k = 3$, $j \in \{1, 2, 3\}$). In contrast, a membership vector in FCM could be represented as follows:

$$\begin{bmatrix} \boldsymbol{\mu}^{(1)} \to 0.1 \\ \boldsymbol{\mu}^{(2)} \to 0.85 \\ \boldsymbol{\mu}^{(3)} \to 0.05 \end{bmatrix}$$

Here, each value falls in the range [0, 1] and represents a probability of membership to the respective cluster centroid. The sum of the memberships for a given sample is equal to 1. Similarly to the k-means algorithm, we can summarize the FCM algorithm in four key steps:

1. Specify the number of k centroids and randomly assign the cluster memberships for each point.

2. Compute the cluster centroids $\boldsymbol{\mu}^{(j)}$, $j \in \{1,\ldots,k\}$.

3. Update the cluster memberships for each point.

4. Repeat steps 2 and 3 until the membership coefficients do not change or a user-defined tolerance or a maximum number of iterations is reached.

The objective function of FCM—we abbreviate it by J_m—looks very similar to the **within cluster sum-squared-error** that we minimize in k-means:

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{m(i,j)} \left\| x^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2 , \; m \in [1, \infty)$$

However, note that the membership indicator $w^{(i,j)}$ is not a binary value as in k-means $w^{(i,j)} \in \{0,1\}$) but a real value that denotes the cluster membership probability $(w^{(i,j)} \in [0,1])$. You also may have noticed that we added an additional exponent to $w^{(i,j)}$; the exponent *m*, any number greater or equal to 1 (typically *m* = 2), is the so-called **fuzziness coefficient** (or simply **fuzzifier**) that controls the degree of **fuzziness**. The larger the value of $\boldsymbol{m}$, the smaller the cluster membership $w^{(i,j)}$ becomes, which leads to fuzzier clusters. The cluster membership probability itself is calculated as follows:

$$
w^{(i,j)} = \left[\sum_{p=1}^{k} \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(p)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}
$$

For example, if we chose three cluster centers as in the previous k-means example, we could calculate the membership of the $\boldsymbol{x}^{(i)}$ sample belonging to the $\mu^{(j)}$ cluster as:

$$
w^{(i,j)} = \left[\left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}
$$

The center $\mu^{(j)}$ of a cluster itself is calculated as the mean of all samples in the cluster weighted by the membership degree of belonging to its own cluster:

$$
\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} w^{m(i,j)} \boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{m(i,j)}}
$$

Just by looking at the equation to calculate the cluster memberships, it is intuitive to say that each iteration in FCM is more expensive than an iteration in k-means. However, FCM typically requires fewer iterations overall to reach convergence. Unfortunately, the FCM algorithm is currently not implemented in scikit-learn. However, it has been found in practice that both k-means and FCM produce very similar clustering outputs, as described in a study by Soumi Ghosh and Sanjay K. Dubey (S. Ghosh and S. K. Dubey. *Comparative Analysis of k-means and Fuzzy c-means Algorithms*. IJACSA, 4:35–38, 2013).

Using the elbow method to find the optimal number of clusters
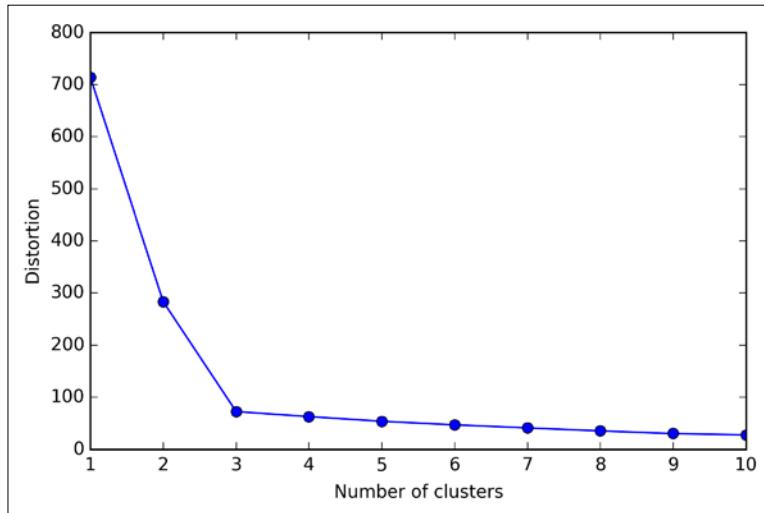
One of the main challenges in unsupervised learning is that we do not know the definitive answer. We don't have the ground truth class labels in our dataset that allow us to apply the techniques that we used in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, in order to evaluate the performance of a supervised model. Thus, in order to quantify the quality of clustering, we need to use intrinsic metrics—such as the within-cluster SSE (distortion) that we discussed earlier in this chapter—to compare the performance of different k-means clusterings. Conveniently, we don't need to compute the within-cluster SSE explicitly as it is already accessible via the `inertia_` attribute after fitting a `KMeans` model:

```
>>> print('Distortion: %.2f' % km.inertia_)
Distortion: 72.48
```

Based on the within-cluster SSE, we can use a graphical tool, the so-called **elbow** method, to estimate the optimal number of clusters *k* for a given task. Intuitively, we can say that, if *k* increases, the distortion will decrease. This is because the samples will be closer to the centroids they are assigned to. The idea behind the elbow method is to identify the value of *k* where the distortion begins to increase most rapidly, which will become more clear if we plot distortion for different values of *k*:

```
>>> distortions = []
>>> for i in range(1, 11):
...     km = KMeans(n_clusters=i,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 random_state=0)
>>>     km.fit(X)
>>>     distortions.append(km.inertia_)
>>> plt.plot(range(1,11), distortions, marker='o')
>>> plt.xlabel('Number of clusters')
>>> plt.ylabel('Distortion')
>>> plt.show()
```

As we can see in the following plot, the *elbow* is located at *k = 3*, which provides evidence that *k = 3* is indeed a good choice for this dataset:

Quantifying the quality of clustering via silhouette plots

Another intrinsic metric to evaluate the quality of a clustering is **silhouette analysis**, which can also be applied to clustering algorithms other than k-means that we will discuss later in this chapter. Silhouette analysis can be used as a graphical tool to plot a measure of how tightly grouped the samples in the clusters are. To calculate the **silhouette coefficient** of a single sample in our dataset, we can apply the following three steps:

1. Calculate the cluster cohesion $a^{(i)}$ as the average distance between a sample $x^{(i)}$ and all other points in the same cluster.

2. Calculate the cluster separation $b^{(i)}$ from the next closest cluster as the average distance between the sample $x^{(i)}$ and all samples in the nearest cluster.

3. Calculate the silhouette $s^{(i)}$ as the difference between cluster cohesion and separation divided by the greater of the two, as shown here:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\left\{b^{(i)}, a^{(i)}\right\}}$$

The silhouette coefficient is bounded in the range -1 to 1. Based on the preceding formula, we can see that the silhouette coefficient is 0 if the cluster separation and cohesion are equal ($b^{(i)} = a^{(i)}$). Furthermore, we get close to an ideal silhouette coefficient of 1 if $b^{(i)} \gg a^{(i)}$, since $b^{(i)}$ quantifies how dissimilar a sample is to other clusters, and $a^{(i)}$ tells us how similar it is to the other samples in its own cluster, respectively.
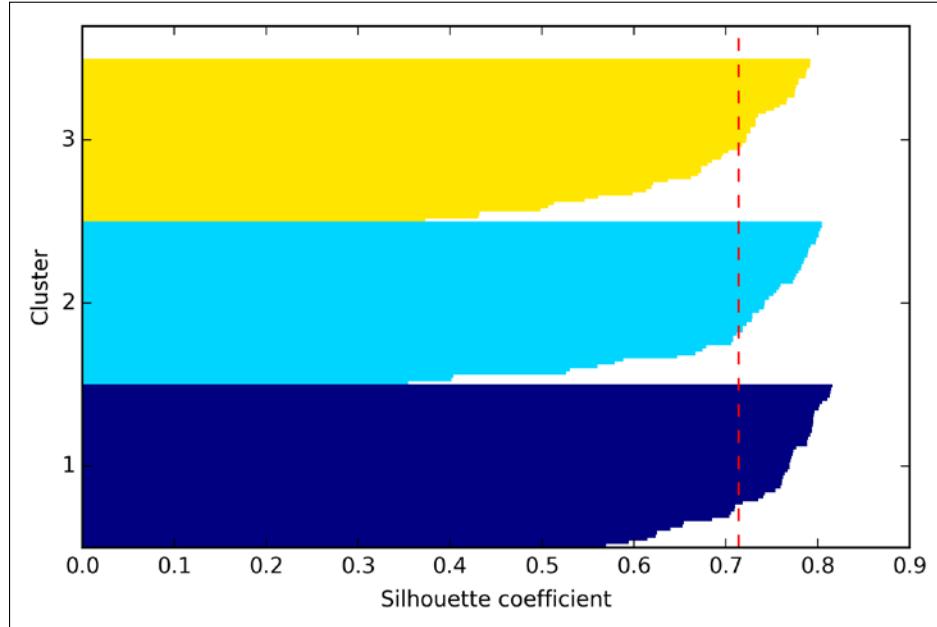
The silhouette coefficient is available as `silhouette_samples` from scikit-learn's `metric` module, and optionally the `silhouette_scores` can be imported. This calculates the average silhouette coefficient across all samples, which is equivalent to `numpy.mean(silhouette_samples(...))`. By executing the following code, we will now create a plot of the silhouette coefficients for a k-means clustering with $k = 3$:

```
>>> km = KMeans(n_clusters=3,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)

>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                      y_km,
...                                      metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...             color="red",
...             linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
```

```
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.show()
```

Through a visual inspection of the silhouette plot, we can quickly scrutinize the sizes of the different clusters and identify clusters that contain *outliers*:

As we can see in the preceding silhouette plot, our silhouette coefficients are not even close to 0, which can be an indicator of a good clustering. Furthermore, to summarize the goodness of our clustering, we added the average silhouette coefficient to the plot (dotted line).

To see how a silhouette plot looks for a relatively *bad* clustering, let's seed the k-means algorithm with two centroids only:

```
>>> km = KMeans(n_clusters=2,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)

>>> plt.scatter(X[y_km==0,0],
...             X[y_km==0,1],
...             s=50, c='lightgreen',
```
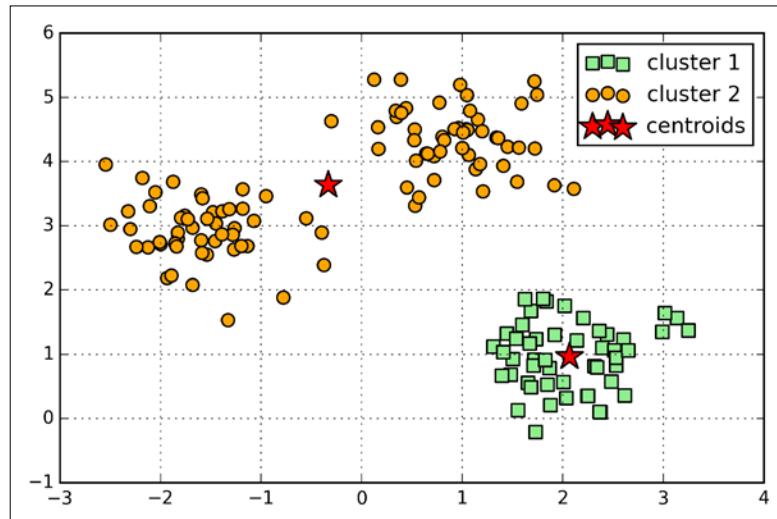
```
...                    marker='s',
...                    label='cluster 1')
>>> plt.scatter(X[y_km==1,0],
...                    X[y_km==1,1],
...                    s=50,
...                    c='orange',
...                    marker='o',
...                    label='cluster 2')
>>> plt.scatter(km.cluster_centers_[:,0],
...                    km.cluster_centers_[:,1],
...                    s=250,
...                    marker='*',
...                    c='red',
...                    label='centroids')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()
```

As we can see in the following scatterplot, one of the centroids falls between two of the three spherical groupings of the sample points. Although the clustering does not look completely terrible, it is suboptimal.

Next we create the silhouette plot to evaluate the results. Please keep in mind that we typically do not have the luxury of visualizing datasets in two-dimensional scatterplots in real-world problems, since we typically work with data in higher dimensions:

```
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
```

```
...                                      y_km,
...                                      metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...             c_silhouette_vals,
...             height=1.0,
...             edgecolor='none',
...             color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.show()
```

As we can see in the resulting plot, the silhouettes now have visibly different lengths and width, which yields further evidence for a suboptimal clustering:
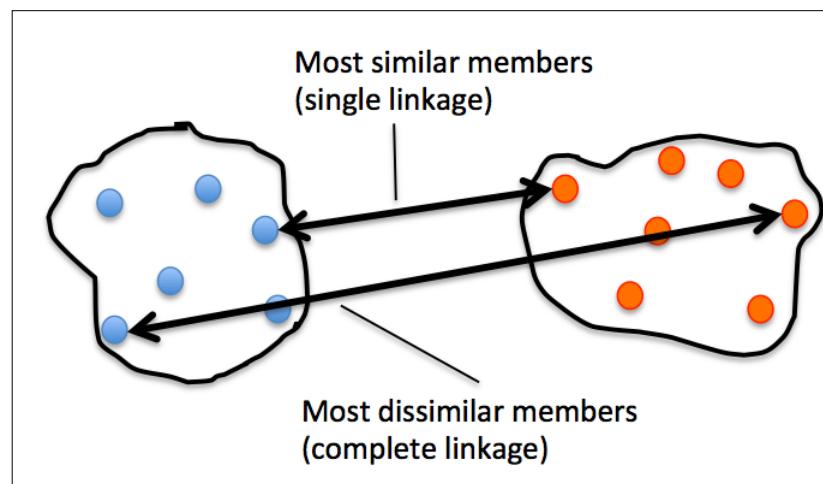
Organizing clusters as a hierarchical tree

In this section, we will take a look at an alternative approach to prototype-based clustering: **hierarchical clustering**. One advantage of hierarchical clustering algorithms is that it allows us to plot **dendrograms** (visualizations of a binary hierarchical clustering), which can help with the interpretation of the results by creating meaningful taxonomies. Another useful advantage of this hierarchical approach is that we do not need to specify the number of clusters upfront.

The two main approaches to hierarchical clustering are **agglomerative** and **divisive** hierarchical clustering. In divisive hierarchical clustering, we start with one cluster that encompasses all our samples, and we iteratively split the cluster into smaller clusters until each cluster only contains one sample. In this section, we will focus on agglomerative clustering, which takes the opposite approach. We start with each sample as an individual cluster and merge the closest pairs of clusters until only one cluster remains.

The two standard algorithms for agglomerative hierarchical clustering are **single linkage** and **complete linkage**. Using single linkage, we compute the distances between the most similar members for each pair of clusters and merge the two clusters for which the distance between the most similar members is the smallest. The complete linkage approach is similar to single linkage but, instead of comparing the most similar members in each pair of clusters, we compare the most dissimilar members to perform the merge. This is shown in the following diagram:

> Other commonly used algorithms for agglomerative hierarchical clustering include **average linkage** and **Ward's linkage**. In average linkage, we merge the cluster pairs based on the minimum average distances between all group members in the two clusters. In Ward's method, those two clusters that lead to the minimum increase of the total within-cluster SSE are merged.

In this section, we will focus on agglomerative clustering using the complete linkage approach. This is an iterative procedure that can be summarized by the following steps:

1. Compute the distance matrix of all samples.
2. Represent each data point as a singleton cluster.
3. Merge the two closest clusters based on the distance of the most dissimilar (distant) members.
4. Update the distance matrix.
5. Repeat steps 2 to 4 until one single cluster remains.

Now we will discuss how to compute the distance matrix (step 1). But first, let's generate some random sample data to work with. The rows represent different observations (IDs 0 to 4), and the columns are the different features (X, Y, Z) of those samples:

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0','ID_1','ID_2','ID_3','ID_4']
>>> X = np.random.random_sample([5,3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

After executing the preceding code, we should now see the following DataFrame containing the randomly generated samples:

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

Performing hierarchical clustering on a distance matrix

To calculate the distance matrix as input for the hierarchical clustering algorithm, we will use the `pdist` function from SciPy's `spatial.distance` submodule:

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...             pdist(df, metric='euclidean')),
...             columns=labels, index=labels)
>>> row_dist
```

Using the preceding code, we calculated the Euclidean distance between each pair of sample points in our dataset based on the features X, Y, and Z. We provided the condensed distance matrix—returned by `pdist`—as input to the `squareform` function to create a symmetrical matrix of the pair-wise distances, as shown here:

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

Next we will apply the complete linkage agglomeration to our clusters using the linkage function from SciPy's cluster.hierarchy submodule, which returns a so-called **linkage matrix**.

However, before we call the linkage function, let's take a careful look at the function documentation:

```
>>> from scipy.cluster.hierarchy import linkage
>>> help(linkage)
[...]
Parameters:
  y : ndarray
    A condensed or redundant distance matrix. A condensed
    distance matrix is a flat array containing the upper
    triangular of the distance matrix. This is the form
    that pdist returns. Alternatively, a collection of m
    observation vectors in n dimensions may be passed as
    an m by n array.

  method : str, optional
    The linkage algorithm to use. See the Linkage Methods
    section below for full descriptions.

  metric : str, optional
    The distance metric to use. See the distance.pdist
    function for a list of valid distance metrics.

 Returns:
  Z : ndarray
    The hierarchical clustering encoded as a linkage matrix.
[...]
```

Based on the function description, we conclude that we can use a condensed distance matrix (upper triangular) from the pdist function as an input attribute. Alternatively, we could also provide the initial data array and use the euclidean metric as a function argument in linkage. However, we should not use the squareform distance matrix that we defined earlier, since it would yield different distance values from those expected. To sum it up, the three possible scenarios are listed here:

- **Incorrect approach**: In this approach, we use the squareform distance matrix. The code is as follows:

```
>>> from scipy.cluster.hierarchy import linkage
>>> row_clusters = linkage(row_dist,
...                        method='complete',
...                        metric='euclidean')
```

- **Correct approach**: In this approach, we use the condensed distance matrix. The code is as follows:

```
>>> row_clusters = linkage(pdist(df, metric='euclidean'),
...                         method='complete')
```

- **Correct approach**: In this approach, we use the input sample matrix. The code is as follows:

```
>>> row_clusters = linkage(df.values,
...                         method='complete',
...                         metric='euclidean')
```

To take a closer look at the clustering results, we can turn them to a pandas `DataFrame` (best viewed in IPython Notebook) as follows:

```
>>> pd.DataFrame(row_clusters,
...       columns=['row label 1',
...                'row label 2',
...                'distance',
...                'no. of items in clust.'],
...       index=['cluster %d' %(i+1) for i in
...              range(row_clusters.shape[0])])
```

As shown in the following table, the linkage matrix consists of several rows where each row represents one merge. The first and second columns denote the most dissimilar members in each cluster, and the third row reports the distance between those members. The last column returns the count of the members in each cluster.
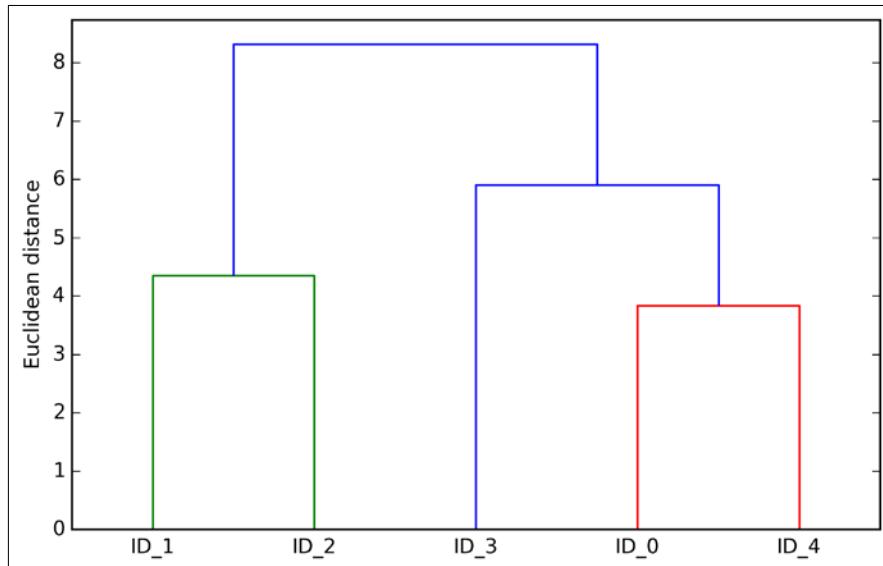
	row label 1	row label 2	distance	no. of items in clust.
cluster 1	0	4	3.835396	2
cluster 2	1	2	4.347073	2
cluster 3	3	5	5.899885	3
cluster 4	6	7	8.316594	5

Now that we have computed the linkage matrix, we can visualize the results in the form of a dendrogram:

```
>>> from scipy.cluster.hierarchy import dendrogram
# make dendrogram black (part 1/2)
# from scipy.cluster.hierarchy import set_link_color_palette
# set_link_color_palette(['black'])
```

```
>>> row_dendr = dendrogram(row_clusters,
...                        labels=labels,
...                        # make dendrogram black (part 2/2)
...                        # color_threshold=np.inf
...                        )
>>> plt.tight_layout()
>>> plt.ylabel('Euclidean distance')
>>> plt.show()
```

If you are executing the preceding code or reading the e-book version of this book, you will notice that the branches in the resulting dendrogram are shown in different colors. The coloring scheme is derived from a list of matplotlib colors that are cycled for the distance thresholds in the dendrogram. For example, to display the dendrograms in black, you can uncomment the respective sections that I inserted in the preceding code.

Such a dendrogram summarizes the different clusters that were formed during the agglomerative hierarchical clustering; for example, we can see that the samples **ID_0** and **ID_4**, followed by **ID_1** and **ID_2**, are the most similar ones based on the Euclidean distance metric.

Attaching dendrograms to a heat map

In practical applications, hierarchical clustering dendrograms are often used in combination with a **heat map**, which allows us to represent the individual values in the sample matrix with a color code. In this section, we will discuss how to attach a dendrogram to a heat map plot and order the rows in the heat map correspondingly.

However, attaching a dendrogram to a heat map can be a little bit tricky, so let's go through this procedure step by step:

1. We create a new `figure` object and define the *x* axis position, *y* axis position, width, and height of the dendrogram via the `add_axes` attribute. Furthermore, we rotate the dendrogram 90 degrees counter-clockwise. The code is as follows:

    ```
    >>> fig = plt.figure(figsize=(8,8), facecolor='white')
    >>> axd = fig.add_axes([0.09,0.1,0.2,0.6])
    >>> row_dendr = dendrogram(row_clusters, orientation='right')
    # note: for matplotlib >= v1.5.1, please use orientation='left'
    ```

2. Next we reorder the data in our initial `DataFrame` according to the clustering labels that can be accessed from the dendrogram object, which is essentially a Python dictionary, via the `leaves` key. The code is as follows:

    ```
    >>> df_rowclust = df.ix[row_dendr['leaves'][::-1]]
    ```

3. Now we construct the heat map from the reordered `DataFrame` and position it right next to the dendrogram:
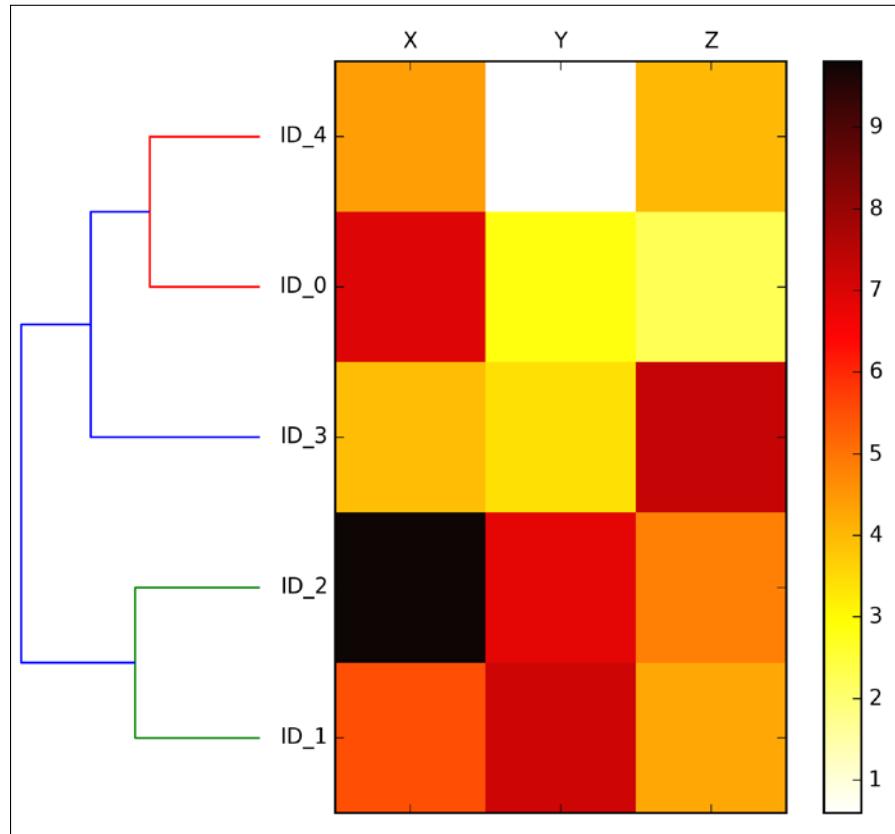
    ```
    >>> axm = fig.add_axes([0.23,0.1,0.6,0.6])
    >>> cax = axm.matshow(df_rowclust,
    ...              interpolation='nearest', cmap='hot_r')
    ```

4. Finally we will modify the aesthetics of the heat map by removing the axis ticks and hiding the axis spines. Also, we will add a color bar and assign the feature and sample names to the *x* and *y* axis tick labels, respectively. The code is as follows:

    ```
    >>> axd.set_xticks([])
    >>> axd.set_yticks([])
    >>> for i in axd.spines.values():
    ...     i.set_visible(False)
    >>> fig.colorbar(cax)
    >>> axm.set_xticklabels([''] + list(df_rowclust.columns))
    >>> axm.set_yticklabels([''] + list(df_rowclust.index))
    >>> plt.show()
    ```

After following the previous steps, the heat map should be displayed with the
dendrogram attached:

As we can see, the row order in the heat map reflects the clustering of the samples
in the dendrogram. In addition to a simple dendrogram, the color-coded values
of each sample and feature in the heat map provide us with a nice summary of
the dataset.

Applying agglomerative clustering via scikit-learn

In this section, we saw how to perform agglomerative hierarchical clustering using SciPy. However, there is also an `AgglomerativeClustering` implementation in scikit-learn, which allows us to choose the number of clusters that we want to return. This is useful if we want to prune the hierarchical cluster tree. By setting the `n_cluster` parameter to `2`, we will now cluster the samples into two groups using the same complete linkage approach based on the Euclidean distance metric as before:

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=2,
...                              affinity='euclidean',
...                              linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print('Cluster labels: %s' % labels)
Cluster labels: [0 1 1 0 0]
```

Looking at the predicted cluster labels, we can see that the first, fourth, and fifth sample (**ID_0**, **ID_3**, and **ID_4**) were assigned to one cluster (`0`), and the samples **ID_1** and **ID_2** were assigned to a second cluster (`1`), which is consistent with the results that we can observe in the dendrogram.

Locating regions of high density via DBSCAN

Although we can't cover the vast number of different clustering algorithms in this chapter, let's at least introduce one more approach to clustering: **Density-based Spatial Clustering of Applications with Noise** (**DBSCAN**). The notion of density in DBSCAN is defined as the number of points within a specified radius ε.
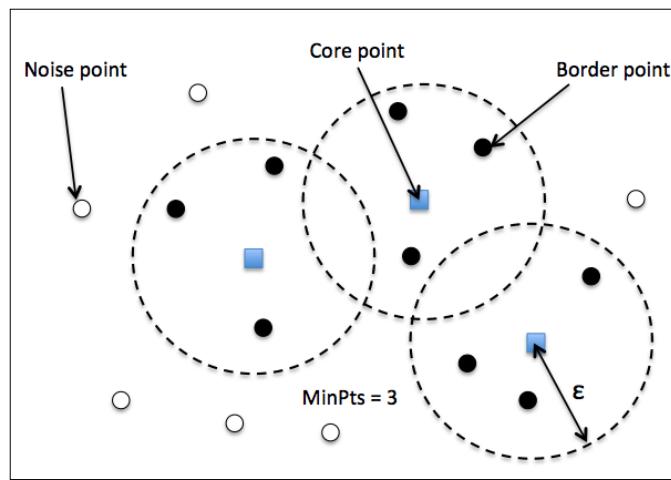
In DBSCAN, a special label is assigned to each sample (point) using the following criteria:

- A point is considered as **core point** if at least a specified number (MinPts) of neighboring points fall within the specified radius ε
- A **border point** is a point that has fewer neighbors than MinPts within ε, but lies within the ε radius of a core point
- All other points that are neither core nor border points are considered as **noise points**

After labeling the points as core, border, or noise points, the DBSCAN algorithm can be summarized in two simple steps:

1. Form a separate cluster for each core point or a connected group of core points (core points are connected if they are no farther away than ε).

2. Assign each border point to the cluster of its corresponding core point.

To get a better understanding of what the result of DBSCAN can look like before jumping to the implementation, let's summarize what you have learned about core points, border points, and noise points in the following figure:
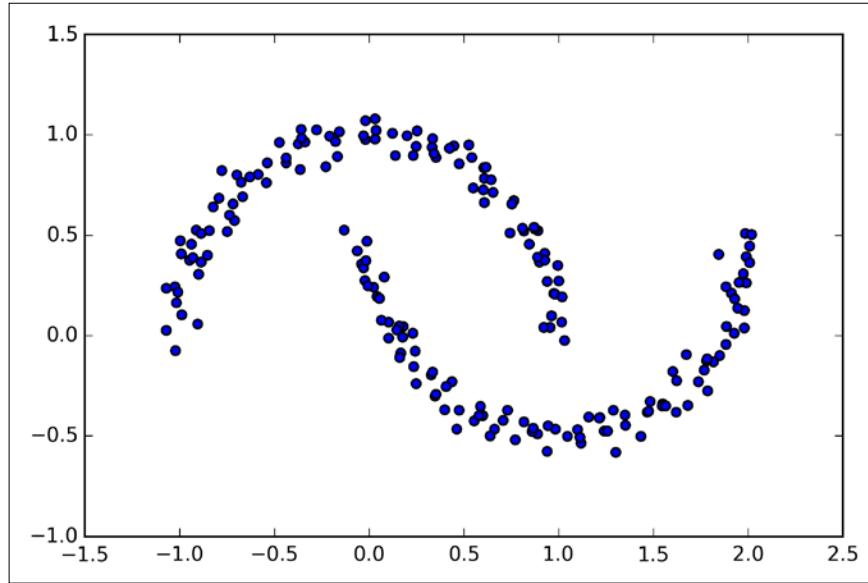
One of the main advantages of using DBSCAN is that it does not assume that the clusters have a spherical shape as in k-means. Furthermore, DBSCAN is different from k-means and hierarchical clustering in that it doesn't necessarily assign each point to a cluster but is capable of removing noise points.

For a more illustrative example, let's create a new dataset of half-moon-shaped structures to compare k-means clustering, hierarchical clustering, and DBSCAN:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=200,
...                     noise=0.05,
...                     random_state=0)
>>> plt.scatter(X[:,0], X[:,1])
>>> plt.show()
```

As we can see in the resulting plot, there are two visible, half-moon-shaped groups consisting of 100 sample points each:

We will start by using the k-means algorithm and complete linkage clustering to see whether one of those previously discussed clustering algorithms can successfully identify the half-moon shapes as separate clusters. The code is as follows:

```
>>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8,3))
>>> km = KMeans(n_clusters=2,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> ax1.scatter(X[y_km==0,0],
...             X[y_km==0,1],
...             c='lightblue',
...             marker='o',
...             s=40,
...             label='cluster 1')
>>> ax1.scatter(X[y_km==1,0],
...             X[y_km==1,1],
...             c='red',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> ax1.set_title('K-means clustering')
>>> ac = AgglomerativeClustering(n_clusters=2,
```
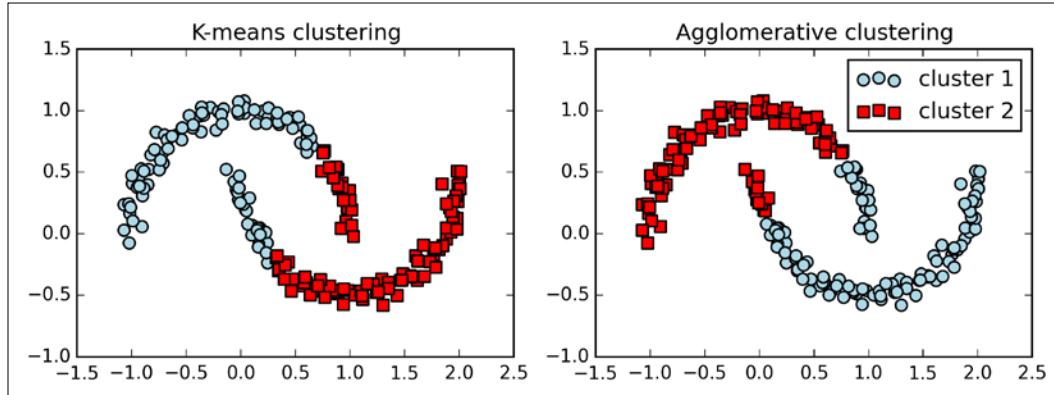
```
...                                      affinity='euclidean',
...                                      linkage='complete')
>>> y_ac = ac.fit_predict(X)
>>> ax2.scatter(X[y_ac==0,0],
...             X[y_ac==0,1],
...             c='lightblue',
...             marker='o',
...             s=40,
...             label='cluster 1')
>>> ax2.scatter(X[y_ac==1,0],
...             X[y_ac==1,1],
...             c='red',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> ax2.set_title('Agglomerative clustering')
>>> plt.legend()
>>> plt.show()
```

Based on the visualized clustering results, we can see that the k-means algorithm is unable to separate the two clusters, and the hierarchical clustering algorithm was challenged by those complex shapes:

Finally, let's try the DBSCAN algorithm on this dataset to see if it can find the two half-moon-shaped clusters using a density-based approach:
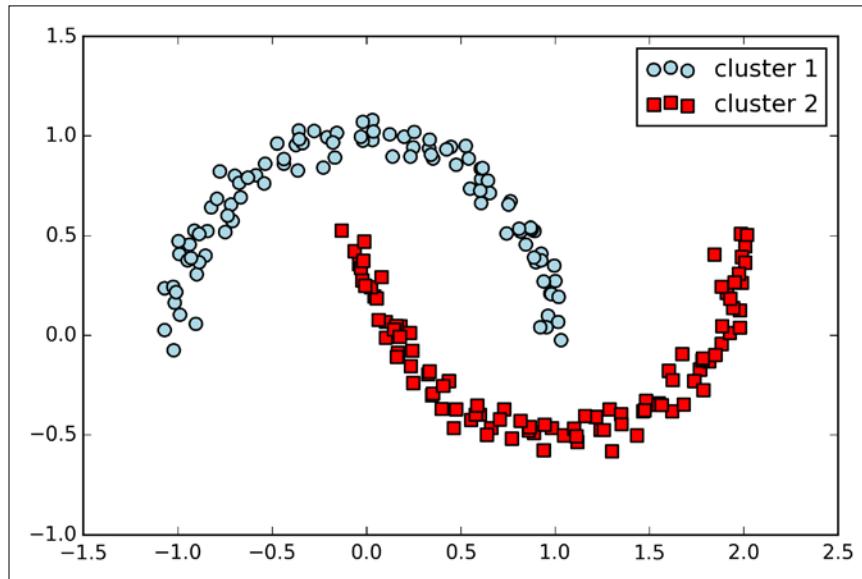
```
>>> from sklearn.cluster import DBSCAN
>>> db = DBSCAN(eps=0.2,
...             min_samples=5,
...             metric='euclidean')
>>> y_db = db.fit_predict(X)
```

```
>>> plt.scatter(X[y_db==0,0],
...             X[y_db==0,1],
...             c='lightblue',
...             marker='o',
...             s=40,
...             label='cluster 1')
>>> plt.scatter(X[y_db==1,0],
...             X[y_db==1,1],
...             c='red',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> plt.legend()
>>> plt.show()
```

The DBSCAN algorithm can successfully detect the half-moon shapes, which highlights one of the strengths of DBSCAN (clustering data of arbitrary shapes)

However, we should also note some of the disadvantages of DBSCAN. With an increasing number of features in our dataset—given a fixed size training set—the negative effect of the *curse of dimensionality* increases. This is especially a problem if we are using the Euclidean distance metric. However, the problem of the *curse of dimensionality* is not unique to DBSCAN; it also affects other clustering algorithms that use the Euclidean distance metric, for example, the k-means and hierarchical clustering algorithms. In addition, we have two hyperparameters in DBSCAN (MinPts and ε) that need to be optimized to yield good clustering results. Finding a good combination of MinPts and ε can be problematic if the density differences in the dataset are relatively large.

> So far, we saw three of the most fundamental categories of clustering algorithms: prototype-based clustering with k-means, agglomerative hierarchical clustering, and density-based clustering via DBSCAN. However, I also want to mention a fourth class of more advanced clustering algorithms that we have not covered in this chapter: **graph-based clustering**. Probably the most prominent members of the graph-based clustering family are **spectral clustering algorithms**. Although there are many different implementations of spectral clustering, they all have in common that they use the eigenvectors of a similarity matrix to derive the cluster relationships. Since spectral clustering is beyond the scope of this book, you can read the excellent tutorial by Ulrike von Luxburg to learn more about this topic (U. Von Luxburg. *A Tutorial on Spectral Clustering*. Statistics and computing, 17(4):395–416, 2007). It is freely available from arXiv at `http://arxiv.org/pdf/0711.0189v1.pdf`.

Note that, in practice, it is not always obvious which algorithm will perform best on a given dataset, especially if the data comes in multiple dimensions that make it hard or impossible to visualize. Furthermore, it is important to emphasize that a successful clustering does not only depend on the algorithm and its hyperparameters. Rather, the choice of an appropriate distance metric and the use of domain knowledge that can help guide the experimental setup can be even more important.

Summary

In this chapter, you learned about three different clustering algorithms that can help us with the discovery of hidden structures or information in data. We started this chapter with a prototype-based approach, k-means, which clusters samples into spherical shapes based on a specified number of cluster centroids. Since clustering is an unsupervised method, we do not enjoy the luxury of ground truth labels to evaluate the performance of a model. Thus, we looked at useful intrinsic performance metrics such as the elbow method or silhouette analysis as an attempt to quantify the quality of clustering.

We then looked at a different approach to clustering: agglomerative hierarchical clustering. Hierarchical clustering does not require specifying the number of clusters upfront, and the result can be visualized in a dendrogram representation, which can help with the interpretation of the results. The last clustering algorithm that we saw in this chapter was DBSCAN, an algorithm that groups points based on local densities and is capable of handling outliers and identifying nonglobular shapes.

After this excursion into the field of unsupervised learning, it is now about time to introduce some of the most exciting machine learning algorithms for supervised learning: multilayer artificial neural networks. After their recent resurgence, neural networks are once again the hottest topic in machine learning research. Thanks to the recently developed deep learning algorithms, neural networks are conceived as state-of-the-art for many complex tasks such as image classification and speech recognition. In *Chapter 12*, *Training Artificial Neural Networks for Image Recognition*, we will construct our own multilayer neural network from scratch. In *Chapter 13*, *Parallelizing Neural Network Training with Theano*, we will introduce powerful libraries that can help us to train complex network architectures most efficiently.

12
Training Artificial Neural Networks for Image Recognition

As you may know, **deep learning** is getting a lot of press and is without any doubt the hottest topic in the machine learning field. Deep learning can be understood as a set of algorithms that were developed to train **artificial neural networks** with many layers most efficiently. In this chapter, you will learn the basic concepts of artificial neural networks so that you will be well equipped to further explore the most exciting areas of research in the machine learning field, as well as the advanced Python-based deep learning libraries that are currently being developed.

The topics that we will cover are as follows:

- Getting a conceptual understanding of multi-layer neural networks
- Training neural networks for image classification
- Implementing the powerful backpropagation algorithm
- Debugging neural network implementations

Modeling complex functions with artificial neural networks
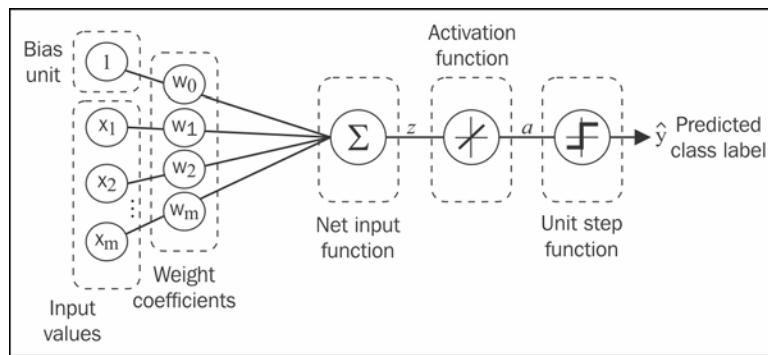
At the beginning of this book, we started our journey through machine learning algorithms with artificial neurons in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. Artificial neurons represent the building blocks of the multi-layer artificial neural networks that we are going to discuss in this chapter. The basic concept behind artificial neural networks was built upon hypotheses and models of how the human brain works to solve complex problem tasks. Although artificial neural networks have gained a lot of popularity in recent years, early studies of neural networks go back to the 1940s when Warren McCulloch and Walter Pitt first described how neurons could work. However, in the decades that followed the first implementation of the **McCulloch-Pitt neuron** model, Rosenblatt's perceptron in the 1950s, many researchers and machine learning practitioners slowly began to lose interest in neural networks since no one had a good solution for training a neural network with multiple layers. Eventually, interest in neural networks was rekindled in 1986 when D.E. Rumelhart, G.E. Hinton, and R.J. Williams were involved in the (re)discovery and popularization of the **backpropagation** algorithm to train neural networks more efficiently, which we will discuss in more detail later in this chapter (Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986). *Learning Representations by Back-propagating Errors*. Nature 323 (6088): 533–536).

During the previous decade, many more major breakthroughs resulted in what we now call deep learning algorithms, which can be used to create **feature detectors** from unlabeled data to pre-train deep neural networks—neural networks that are composed of many layers. Neural networks are a hot topic not only in academic research, but also in big technology companies such as Facebook, Microsoft, and Google who invest heavily in artificial neural networks and deep learning research. As of today, complex neural networks powered by deep learning algorithms are considered as state-of-the-art when it comes to complex problem solving such as image and voice recognition. Popular examples of the products in our everyday life that are powered by deep learning are Google's image search and Google Translate, an application for smartphones that can automatically recognize text in images for real-time translation into 20 languages (`http://googleresearch.blogspot.com/2015/07/how-google-translate-squeezes-deep.html`).

Many more exciting applications of deep neural networks are under active development at major tech companies, for example, Facebook's DeepFace for tagging images (Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. *DeepFace: Closing the gap to human-level performance in face verification*. In Computer Vision and Pattern Recognition CVPR, 2014 IEEE Conference, pages 1701–1708) and Baidu's DeepSpeech, which is able to handle voice queries in Mandarin (A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. *DeepSpeech: Scaling up end-to-end speech recognition*. arXiv preprint arXiv:1412.5567, 2014). In addition, the pharmaceutical industry recently started to use deep learning techniques for drug discovery and toxicity prediction, and research has shown that these novel techniques substantially exceed the performance of traditional methods for virtual screening (T. Unterthiner, A. Mayr, G. Klambauer, and S. Hochreiter. *Toxicity prediction using deep learning*. arXiv preprint arXiv:1503.01445, 2015).

Single-layer neural network recap

This chapter is all about multi-layer neural networks, how they work, and how to train them to solve complex problems. However, before we dig deeper into a particular multi-layer neural network architecture, let's briefly reiterate some of the concepts of single-layer neural networks that we introduced in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, namely, the **ADAptive LInear NEuron** (**Adaline**) algorithm that is shown in the following figure:

In *Chapter 2*, *Training Machine Learning Algorithms for Classification*, we implemented the Adaline algorithm to perform binary classification, and we used a **gradient descent** optimization algorithm to learn the weight coefficients of the model. In every **epoch** (pass over the training set), we updated the weight vector w using the following update rule:

$$w := w + \Delta w, \text{ where } \Delta w = -\eta \nabla J(w)$$

In other words, we computed the gradient based on the whole training set and updated the weights of the model by taking a step into the opposite direction of the gradient $\nabla J(w)$. In order to find the optimal weights of the model, we optimized an objective function that we defined as the **Sum of Squared Errors** (**SSE**) cost function $J(w)$. Furthermore, we multiplied the gradient by a factor, the **learning rate** η, which we chose carefully to balance the speed of learning against the risk of overshooting the global minimum of the cost function.

In gradient descent optimization, we updated all weights simultaneously after each epoch, and we defined the partial derivative for each weight w_j in the weight vector w as follows:

$$\frac{\partial}{\partial w_j} J(w) = \sum_i \left(y^{(i)} - a^{(i)} \right) x_j^{(i)}$$

Here $y^{(i)}$ is the target class label of a particular sample $x^{(i)}$, and $a^{(i)}$ is the **activation** of the neuron, which is a linear function in the special case of Adaline. Furthermore, we defined the *activation function* $\phi(\cdot)$ as follows:
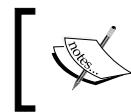
$$\phi(z) = z = a$$

Here, the net input z is a linear combination of the weights that are connecting the input to the output layer:

$$z = \sum_j w_j x_j = w^T x$$

While we used the activation $\phi(z)$ to compute the gradient update, we implemented a **threshold function** (Heaviside function) to squash the continuous-valued output into binary class labels for prediction:

$$\hat{y} = \begin{cases} 1 & \text{if } g(z) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$
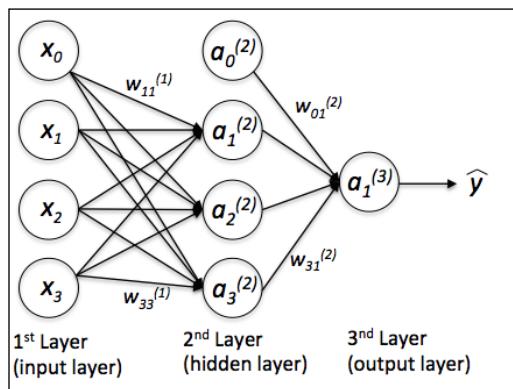
Note that although Adaline consists of two layers, one input layer and one output layer, it is called a single-layer network because of its single link between the input and output layers.

Introducing the multi-layer neural network architecture

In this section, we will see how to connect multiple single neurons to a **multi-layer feedforward neural network**; this special type of network is also called a **multi-layer perceptron** (**MLP**). The following figure explains the concept of an MLP consisting of three layers: one input layer, one **hidden layer**, and one output layer. The units in the hidden layer are fully connected to the input layer, and the output layer is fully connected to the hidden layer, respectively. If such a network has more than one hidden layer, we also call it a *deep* artificial neural network.

We could add an arbitrary number of hidden layers to the MLP to create deeper network architectures. Practically, we can think of the number of layers and units in a neural network as additional **hyperparameters** that we want to optimize for a given problem task using the cross-validation that we discussed in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

However, the error gradients that we will calculate later via backpropagation would become increasingly small as more layers are added to a network. This *vanishing gradient* problem makes the model learning more challenging. Therefore, special algorithms have been developed to pretrain such deep neural network structures, which is called *deep learning*.

As shown in the preceding figure, we denote the ith activation unit in the lth layer as $a_i^{(l)}$, and the activation units $a_0^{(1)}$ and $a_0^{(2)}$ are the **bias units**, respectively, which we set equal to 1. The activation of the units in the input layer is just its input plus the bias unit:

$$a^{(1)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$
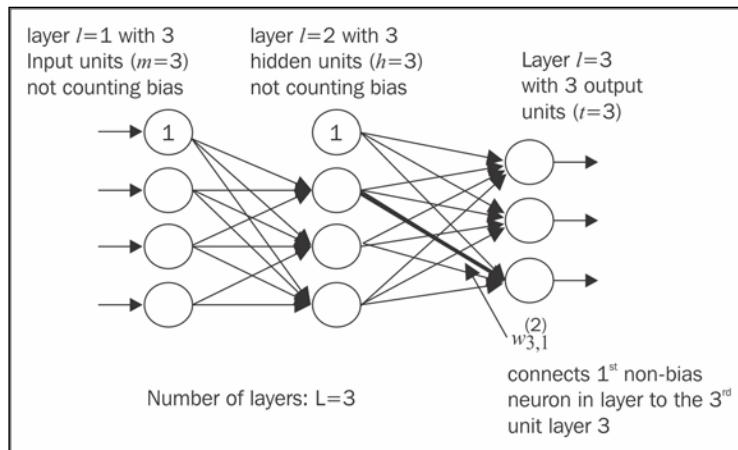
Each unit in layer l is connected to all units in layer $l+1$ via a weight coefficient. For example, the connection between the kth unit in layer l to the jth unit in layer $l+1$ would be written as $w_{j,k}^{(l)}$. Please note that the superscript i in $x_m^{(i)}$ stands for the ith sample, not the ith layer. In the following paragraphs, we will often omit the superscript i for clarity.

While one unit in the output layer would suffice for a binary classification task, we saw a more general form of a neural network in the preceding figure, which allows us to perform multi-class classification via a generalization of the **One-vs-All** (**OvA**) technique. To better understand how this works, remember the **one-hot** representation of categorical variables that we introduced in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*. For example, we would encode the three class labels in the familiar Iris dataset (0=Setosa, 1=Versicolor, 2=Virginica) as follows:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

This one-hot vector representation allows us to tackle classification tasks with an arbitrary number of unique class labels present in the training set.

If you are new to neural network representations, the terminology around the indices (subscripts and superscripts) may look a little bit confusing at first. You may wonder why we wrote $w_{j,k}^{(l)}$ and not $w_{k,j}^{(l)}$ to refer to the weight coefficient that connects the kth unit in layer l to the jth unit in layer $l+1$. What may seem a little bit quirky at first will make much more sense in later sections when we vectorize the neural network representation. For example, we will summarize the weights that connect the input and hidden layer by a matrix $\boldsymbol{W}^{(1)} \in \mathbb{R}^{h \times [m+1]}$, where h is the number of hidden units and $m+1$ is the number of input units plus bias unit. Since it is important to internalize this notation to follow the concepts later in this chapter, let's summarize what we just discussed in a descriptive illustration of a simplified 3-4-3 multi-layer perceptron:

Activating a neural network via forward propagation

In this section, we will describe the process of **forward propagation** to calculate the output of an MLP model. To understand how it fits into the context of learning an MLP model, let's summarize the MLP learning procedure in three simple steps:

1. Starting at the input layer, we forward propagate the patterns of the training data through the network to generate an output.

2. Based on the network's output, we calculate the error that we want to minimize using a cost function that we will describe later.

3. We backpropagate the error, find its derivative with respect to each weight in the network, and update the model.

Finally, after repeating the steps for multiple epochs and learning the weights of the MLP, we use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels in the one-hot representation, which we described in the previous section.

Now, let's walk through the individual steps of forward propagation to generate an output from the patterns in the training data. Since each unit in the hidden layer is connected to all units in the input layers, we first calculate the activation $a_1^{(2)}$ as follows:
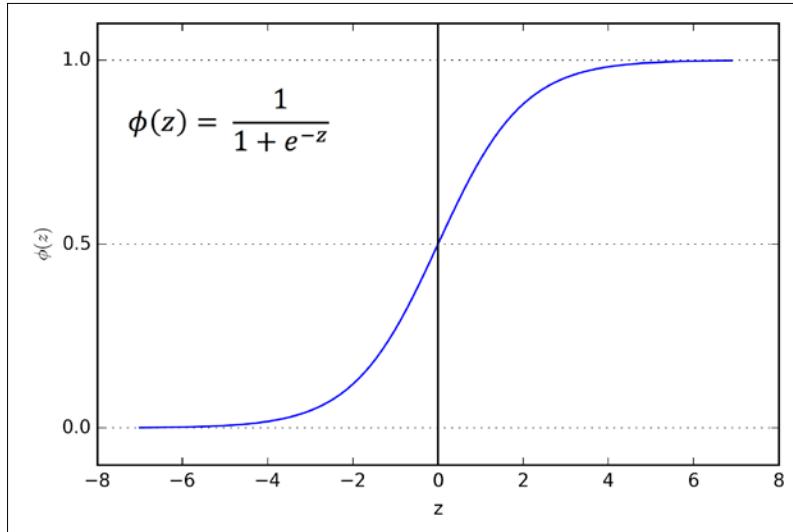
$$z_1^{(2)} = a_0^{(1)} w_{1,0}^{(1)} + a_1^{(1)} w_{1,1}^{(1)} + \cdots + a_m^{(1)} w_{1,m}^{(1)}$$

$$a_1^{(2)} = \phi\left(z_1^{(2)}\right)$$

Here, $z_1^{(2)}$ is the net input and $\phi(\cdot)$ is the activation function, which has to be differentiable to learn the weights that connect the neurons using a gradient-based approach. To be able to solve complex problems such as image classification, we need nonlinear activation functions in our MLP model, for example, the **sigmoid** (**logistic**) activation function that we used in **logistic regression** in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

As we can remember, the sigmoid function is an S-shaped curve that maps the net input z onto a logistic distribution in the range 0 to 1, which cuts the y axis at $z=0$, as shown in the following graph:
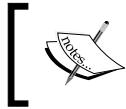
The MLP is a typical example of a *feedforward* artificial neural network. The term *feedforward* refers to the fact that each layer serves as the input to the next layer without loops, in contrast to *recurrent neural networks*, an architecture that we will discuss later in this chapter. The term multi-layer perceptron may sound a little bit confusing, since the artificial neurons in this network architecture are typically sigmoid units, not *perceptrons*. Intuitively, we can think of the neurons in the MLP as logistic regression units that return values in the continuous range between 0 and 1.

For purposes of code efficiency and readability, we will now write the activation in a more compact form using the concepts of basic linear algebra, which will allow us to **vectorize** our code implementation via NumPy rather than writing multiple nested and expensive Python `for` loops:

$$\mathbf{z}^{(2)} = \boldsymbol{W}^{(1)} \boldsymbol{a}^{(1)}$$

$$\boldsymbol{a}^{(2)} = \phi\left(\mathbf{z}^{(2)}\right)$$

> Everywhere you read *h* on this page, you can think of *h* as *h+1* to include the bias unit (and in order to get the dimensions right).

Here, $a^{(1)}$ is our $[m+1] \times 1$ dimensional feature vector of a sample $x^{(i)}$ plus bias unit. $W^{(1)}$ is an $h \times [m+1]$ dimensional weight matrix where h is the number of hidden units in our neural network. After matrix-vector multiplication, we obtain the $h \times 1$ dimensional net input vector $z^{(2)}$ to calculate the activation $a^{(2)}$ (where $a^{(2)} \in \mathbb{R}^{h \times 1}$). Furthermore, we can generalize this computation to all n samples in the training set:

$$Z^{(2)} = W^{(1)} \left[A^{(1)} \right]^T$$

Here, $A^{(1)}$ is now an $n \times [m+1]$ matrix, and the matrix-matrix multiplication will result in a $h \times n$ dimensional net input matrix $z^{(2)}$. Finally, we apply the activation function $\phi(\cdot)$ to each value in the net input matrix to get the $h \times n$ activation matrix $A^{(2)}$ for the next layer (here, output layer):

$$A^{(2)} = \phi\left(Z^{(2)} \right)$$

Similarly, we can rewrite the activation of the output layer in the vectorized form:

$$Z^{(3)} = W^{(2)} A^{(2)}$$

Here, we multiply the $t \times h$ matrix $W^{(2)}$ (t is the number of output units) by the $h \times n$ dimensional matrix $A^{(2)}$ to obtain the $t \times n$ dimensional matrix $Z^{(3)}$ (the columns in this matrix represent the outputs for each sample).

Lastly, we apply the sigmoid activation function to obtain the continuous valued output of our network:

$$A^{(3)} = \phi\left(Z^{(3)} \right), \ A^{(3)} \in \mathbb{R}^{t \times n}$$

Classifying handwritten digits

In the previous section, we covered a lot of the theory around neural networks, which can be a little bit overwhelming if you are new to this topic. Before we continue with the discussion of the algorithm for learning the weights of the MLP model, backpropagation, let's take a short break from the theory and see a neural network in action.

> Neural network theory can be quite complex, thus I want to recommend two additional resources that cover some of the concepts that we discuss in this chapter in more detail:
>
> T. Hastie, J. Friedman, and R. Tibshirani. *The Elements of Statistical Learning*, Volume 2. Springer, 2009.
>
> C. M. Bishop et al. *Pattern Recognition and Machine Learning*, Volume 1. Springer New York, 2006.

In this section, we will train our first multi-layer neural network to classify handwritten digits from the popular **MNIST** dataset (short for **Mixed National Institute of Standards and Technology** database) that has been constructed by Yann LeCun et al. and serves as a popular benchmark dataset for machine learning algorithms (Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. *Gradient-based Learning Applied to Document Recognition*. Proceedings of the IEEE, 86(11):2278-2324, November 1998).

Obtaining the MNIST dataset

The MNIST dataset is publicly available at `http://yann.lecun.com/exdb/mnist/` and consists of the following four parts:

- **Training set images**: `train-images-idx3-ubyte.gz` (9.9 MB, 47 MB unzipped, and 60,000 samples)
- **Training set labels**: `train-labels-idx1-ubyte.gz` (29 KB, 60 KB unzipped, and 60,000 labels)
- **Test set images**: `t10k-images-idx3-ubyte.gz` (1.6 MB, 7.8 MB, unzipped and 10,000 samples)
- **Test set labels**: `t10k-labels-idx1-ubyte.gz` (5 KB, 10 KB unzipped, and 10,000 labels)

The MNIST dataset was constructed from two datasets of the US **National Institute of Standards and Technology** (**NIST**). The training set consists of handwritten digits from 250 different people, 50 percent high school students, and 50 percent employees from the Census Bureau. Note that the test set contains handwritten digits from different people following the same split.

After downloading the files, I recommend unzipping the files using the Unix/Linux gzip tool from the command line terminal for efficiency using the following command in your local MNIST download directory:

```
gzip *ubyte.gz -d
```

Alternatively, you could use your favorite unzipping tool if you are working with a machine running on Microsoft Windows. The images are stored in byte format, and we will read them into NumPy arrays that we will use to train and test our MLP implementation:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                                % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                                % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                   dtype=np.uint8).reshape(len(labels), 784)

    return images, labels
```

The `load_mnist` function returns two arrays, the first being an $n \times m$ dimensional NumPy array (`images`), where n is the number of samples and m is the number of features. The training dataset consists of 60,000 training digits and the test set contains 10,000 samples, respectively. The images in the MNIST dataset consist of 28×28 pixels, and each pixel is represented by a gray scale intensity value. Here, we unroll the 28×28 pixels into 1D row vectors, which represent the rows in our image array (784 per row or image). The second array (`labels`) returned by the `load_mnist` function contains the corresponding target variable, the class labels (integers 0-9) of the handwritten digits.

The way we read in the image might seem a little bit strange at first:

```
magic, n = struct.unpack('>II', lbpath.read(8))
labels = np.fromfile(lbpath, dtype=np.int8)
```

To understand how these two lines of code work, let's take a look at the dataset description from the MNIST website:

[offset]	*[type]*	*[value]*	*[description]*
0000	*32 bit integer*	*0x00000801(2049)*	*magic number (MSB first)*
0004	*32 bit integer*	*60000*	*number of items*
0008	*unsigned byte*	*??*	*label*
0009	*unsigned byte*	*??*	*label*
....			
xxxx	*unsigned byte*	*??*	*label*

Using the two lines of the preceding code, we first read in the *magic number*, which is a description of the file protocol as well as the *number of items* (*n*) from the file buffer before we read the following bytes into a NumPy array using the `fromfile` method. The `fmt` parameter value `>II` that we passed as an argument to `struct.unpack` has two parts:

- `>`: This is big-endian (defines the order in which a sequence of bytes is stored); if you are unfamiliar with the terms *big-endian* and *small-endian*, you can find an excellent article about *Endianness* on Wikipedia (`https://en.wikipedia.org/wiki/Endianness`).
- `I`: This is an unsigned integer.

By executing the following code, we will now load the 60,000 training instances as well as the 10,000 test samples from the `mnist` directory where we unzipped the MNIST dataset:
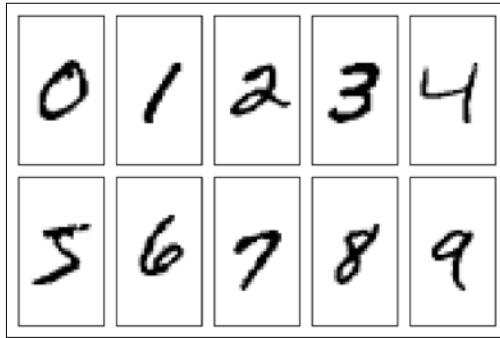
```
>>> X_train, y_train = load_mnist('mnist', kind='train')
>>> print('Rows: %d, columns: %d'
...        % (X_train.shape[0], X_train.shape[1]))
Rows: 60000, columns: 784

>>> X_test, y_test = load_mnist('mnist', kind='t10k')
>>> print('Rows: %d, columns: %d'
...        % (X_test.shape[0], X_test.shape[1]))
Rows: 10000, columns: 784
```

To get a idea what the images in MNIST look like, let's visualize examples of the digits 0-9 after reshaping the 784-pixel vectors from our feature matrix into the original 28 × 28 image that we can plot via matplotlib's `imshow` function:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True,
sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X_train[y_train == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```
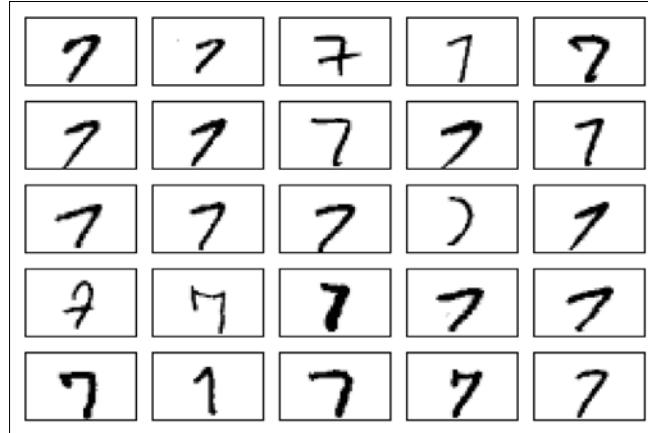
We should now see a plot of the 2×5 subfigures showing a representative image of each unique digit:

In addition, let's also plot multiple examples of the same digit to see how different those handwriting examples really are:

```
>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = X_train[y_train == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code, we should now see the first 25 variants of the digit 7.

Optionally, we can save the MNIST image data and labels as CSV files to open them in programs that do not support their special byte format. However, we should be aware that the CSV file format will take up substantially more space on your local drive, as listed here:

- `train_img.csv`: 109.5 MB
- `train_labels.csv`: 120 KB
- `test_img.csv`: 18.3 MB
- `test_labels.csv`: 20 KB

If we decide to save those CSV files, we can execute the following code in our Python session after loading the MNIST data into NumPy arrays:

```
>>> np.savetxt('train_img.csv', X_train,
...            fmt='%i', delimiter=',')
>>> np.savetxt('train_labels.csv', y_train,
...            fmt='%i', delimiter=',')
>>> np.savetxt('test_img.csv', X_test,
...            fmt='%i', delimiter=',')
>>> np.savetxt('test_labels.csv', y_test,
...            fmt='%i', delimiter=',')
```

Once we have saved the CSV files, we can load them back into Python using NumPy's `genfromtxt` function:

```
>>> X_train = np.genfromtxt('train_img.csv',
...                          dtype=int, delimiter=',')
>>> y_train = np.genfromtxt('train_labels.csv',
...                          dtype=int, delimiter=',')
>>> X_test = np.genfromtxt('test_img.csv',
...                          dtype=int, delimiter=',')
>>> y_test = np.genfromtxt('test_labels.csv',
...                          dtype=int, delimiter=',')
```

However, it will take substantially longer to load the MNIST data from the CSV files, thus I recommend you stick to the original byte format if possible.

Implementing a multi-layer perceptron

In this subsection, we will now implement the code of an MLP with one input, one hidden, and one output layer to classify the images in the MNIST dataset. I have tried to keep the code as simple as possible. However, it may seem a little bit complicated at first, and I encourage you to download the sample code for this chapter from the Packt Publishing website, where you can find this MLP implementation annotated with comments and syntax highlighting for better readability. If you are not running the code from the accompanying IPython notebook, I recommend you copy it into a Python script file in your current working directory, for example, `neuralnet.py`, which you can then import into your current Python session via the following command:

from neuralnet import NeuralNetMLP

The code will contain parts that we have not talked about yet, such as the backpropagation algorithm, but most of the code should look familiar to you based on the Adaline implementation in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, and the discussion of forward propagation in earlier sections. Do not worry if not all of the code makes immediate sense to you; we will follow up on certain parts later in this chapter. However, going over the code at this stage can make it easier to follow the theory later.

```
import numpy as np
from scipy.special import expit
import sys

class NeuralNetMLP(object):
    def __init__(self, n_output, n_features, n_hidden=30,
                 l1=0.0, l2=0.0, epochs=500, eta=0.001,
```

```
            alpha=0.0, decrease_const=0.0, shuffle=True,
            minibatches=1, random_state=None):
    np.random.seed(random_state)
    self.n_output = n_output
    self.n_features = n_features
    self.n_hidden = n_hidden
    self.w1, self.w2 = self._initialize_weights()
    self.l1 = l1
    self.l2 = l2
    self.epochs = epochs
    self.eta = eta
    self.alpha = alpha
    self.decrease_const = decrease_const
    self.shuffle = shuffle
    self.minibatches = minibatches

def _encode_labels(self, y, k):
    onehot = np.zeros((k, y.shape[0]))
    for idx, val in enumerate(y):
        onehot[val, idx] = 1.0
    return onehot

def _initialize_weights(self):
    w1 = np.random.uniform(-1.0, 1.0,
                size=self.n_hidden*(self.n_features + 1))
    w1 = w1.reshape(self.n_hidden, self.n_features + 1)
    w2 = np.random.uniform(-1.0, 1.0,
                size=self.n_output*(self.n_hidden + 1))
    w2 = w2.reshape(self.n_output, self.n_hidden + 1)
    return w1, w2

def _sigmoid(self, z):
    # expit is equivalent to 1.0/(1.0 + np.exp(-z))
    return expit(z)

def _sigmoid_gradient(self, z):
    sg = self._sigmoid(z)
    return sg * (1 - sg)

def _add_bias_unit(self, X, how='column'):
    if how == 'column':
        X_new = np.ones((X.shape[0], X.shape[1]+1))
        X_new[:, 1:] = X
    elif how == 'row':
```

```
        X_new = np.ones((X.shape[0]+1, X.shape[1]))
        X_new[1:, :] = X
    else:
        raise AttributeError('`how` must be `column` or `row`')
    return X_new

def _feedforward(self, X, w1, w2):
    a1 = self._add_bias_unit(X, how='column')
    z2 = w1.dot(a1.T)
    a2 = self._sigmoid(z2)
    a2 = self._add_bias_unit(a2, how='row')
    z3 = w2.dot(a2)
    a3 = self._sigmoid(z3)
    return a1, z2, a2, z3, a3

def _L2_reg(self, lambda_, w1, w2):
    return (lambda_/2.0) * (np.sum(w1[:, 1:] ** 2)\
            + np.sum(w2[:, 1:] ** 2))

def _L1_reg(self, lambda_, w1, w2):
    return (lambda_/2.0) * (np.abs(w1[:, 1:]).sum()\
            + np.abs(w2[:, 1:]).sum())

def _get_cost(self, y_enc, output, w1, w2):
    term1 = -y_enc * (np.log(output))
    term2 = (1 - y_enc) * np.log(1 - output)
    cost = np.sum(term1 - term2)
    L1_term = self._L1_reg(self.l1, w1, w2)
    L2_term = self._L2_reg(self.l2, w1, w2)
    cost = cost + L1_term + L2_term
    return cost

def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):
    # backpropagation
    sigma3 = a3 - y_enc
    z2 = self._add_bias_unit(z2, how='row')
    sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)
    sigma2 = sigma2[1:, :]
    grad1 = sigma2.dot(a1)
    grad2 = sigma3.dot(a2.T)

    # regularize
    grad1[:, 1:] += (w1[:, 1:] * (self.l1 + self.l2))
```

```
        grad2[:, 1:] += (w2[:, 1:] * (self.l1 + self.l2))

        return grad1, grad2

    def predict(self, X):
        a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)
        y_pred = np.argmax(z3, axis=0)
        return y_pred

    def fit(self, X, y, print_progress=False):
        self.cost_ = []
        X_data, y_data = X.copy(), y.copy()
        y_enc = self._encode_labels(y, self.n_output)

        delta_w1_prev = np.zeros(self.w1.shape)
        delta_w2_prev = np.zeros(self.w2.shape)

        for i in range(self.epochs):

            # adaptive learning rate
            self.eta /= (1 + self.decrease_const*i)

            if print_progress:
                sys.stderr.write(
                        '\rEpoch: %d/%d' % (i+1, self.epochs))
                sys.stderr.flush()

            if self.shuffle:
                idx = np.random.permutation(y_data.shape[0])
                X_data, y_enc = X_data[idx], y_enc[:,idx]

            mini = np.array_split(range(
                        y_data.shape[0]), self.minibatches)
            for idx in mini:

                # feedforward
                a1, z2, a2, z3, a3 = self._feedforward(
                                    X_data[idx], self.w1, self.w2)
                cost = self._get_cost(y_enc=y_enc[:, idx],
                                        output=a3,
                                        w1=self.w1,
                                        w2=self.w2)
                self.cost_.append(cost)
```

```
                    # compute gradient via backpropagation
                    grad1, grad2 = self._get_gradient(a1=a1, a2=a2,
                                                      a3=a3, z2=z2,
                                                      y_enc=y_enc[:, idx],
                                                      w1=self.w1,
                                                      w2=self.w2)


                    # update weights
                    delta_w1, delta_w2 = self.eta * grad1,\
                                         self.eta * grad2
                    self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
                    self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
                    delta_w1_prev, delta_w2_prev = delta_w1, delta_w2


        return self
```

Now, let's initialize a new 784-50-10 MLP, a neural network with 784 input units (n_features), 50 hidden units (n_hidden), and 10 output units (n_output):

```
>>> nn = NeuralNetMLP(n_output=10,
...                   n_features=X_train.shape[1],
...                   n_hidden=50,
...                   l2=0.1,
...                   l1=0.0,
...                   epochs=1000,
...                   eta=0.001,
...                   alpha=0.001,
...                   decrease_const=0.00001,
...                   shuffle=True,
...                   minibatches=50,
...                   random_state=1)
```

As you may have noticed, by going over our preceding MLP implementation, we also implemented some additional features, which are summarized here:

- l2: The λ parameter for L2 regularization to decrease the degree of overfitting; equivalently, l1 is the λ parameter for L1 regularization.
- epochs: The number of passes over the training set.

- `eta`: The learning rate η.

- `alpha`: A parameter for momentum learning to add a factor of the previous gradient to the weight update for faster learning $\Delta w_t = \eta \nabla J(w_t) + \alpha \Delta w_{t-1}$ (where t is the current time step or epoch).

- `decrease_const`: The decrease constant d for an adaptive learning rate n that decreases over time for better convergence $\eta / 1 + t \times d$.

- `shuffle`: Shuffling the training set prior to every epoch to prevent the algorithm from getting stuck in cycles.

- `Minibatches`: Splitting of the training data into k mini-batches in each epoch. The gradient is computed for each mini-batch separately instead of the entire training data for faster learning.
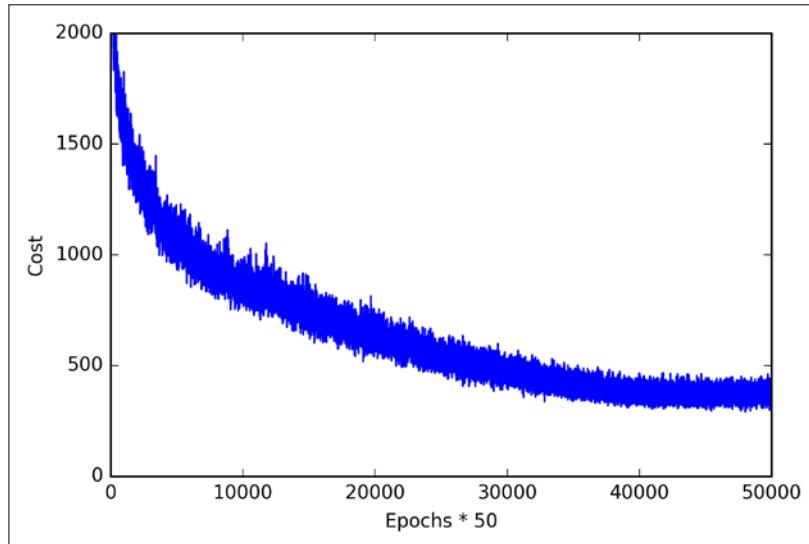
Next, we train the MLP using 60,000 samples from the already shuffled MNIST training dataset. Before you execute the following code, please note that training the neural network may take 10-30 minutes on standard desktop computer hardware:

```
>>> nn.fit(X_train, y_train, print_progress=True)
Epoch: 1000/1000
```

Similar to our previous Adaline implementation, we save the cost for each epoch in a `cost_` list that we can now visualize, making sure that the optimization algorithm reached convergence. Here, we only plot every 50th step to account for the 50 mini-batches (50 mini-batches × 1000 epochs). The code is as follows:

```
>>> plt.plot(range(len(nn.cost_)), nn.cost_)
>>> plt.ylim([0, 2000])
>>> plt.ylabel('Cost')
>>> plt.xlabel('Epochs * 50')
>>> plt.tight_layout()
>>> plt.show()
```

As we see in the following plot, the graph of the cost function looks very noisy. This is due to the fact that we trained our neural network with mini-batch learning, a variant of stochastic gradient descent.
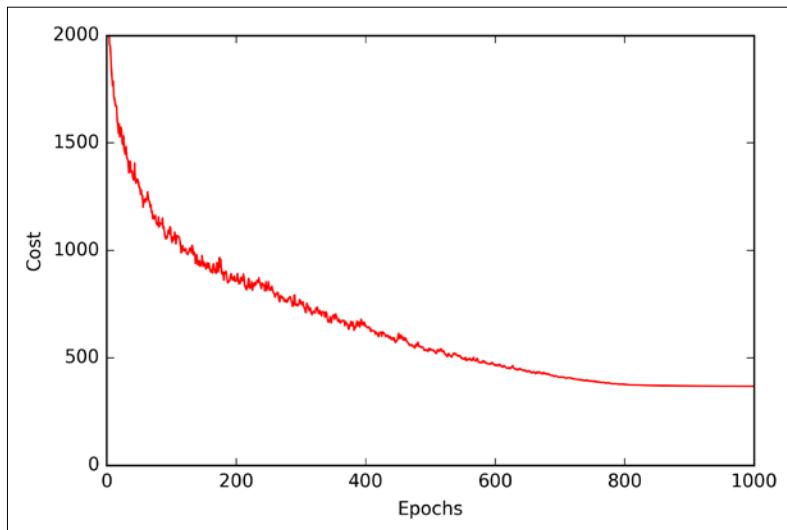
Although we can already see in the plot that the optimization algorithm converged after approximately 800 epochs (*40,000/50 = 800*), let's plot a smoother version of the cost function against the number of epochs by averaging over the mini-batch intervals. The code is as follows:

```
>>> batches = np.array_split(range(len(nn.cost_)), 1000)
>>> cost_ary = np.array(nn.cost_)
>>> cost_avgs = [np.mean(cost_ary[i]) for i in batches]

>>> plt.plot(range(len(cost_avgs)),
...          cost_avgs,
...          color='red')
>>> plt.ylim([0, 2000])
>>> plt.ylabel('Cost')
>>> plt.xlabel('Epochs')
>>> plt.tight_layout()
>>> plt.show()
```

The following plot gives us a clearer picture indicating that the training algorithm converged shortly after the 800th epoch:

Now, let's evaluate the performance of the model by calculating the prediction accuracy:

```
>>> y_train_pred = nn.predict(X_train)
>>> acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (acc * 100))
Training accuracy: 97.59%
```

As we can see, the model classifies most of the training digits correctly, but how does it generalize to data that it has not seen before? Let's calculate the accuracy on 10,000 images in the test dataset:

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
>>> print('Test accuracy: %.2f%%' % (acc * 100))
Test accuracy: 95.62%
```

Based on the small discrepancy between training and test accuracy, we can conclude that the model only slightly overfits the training data. To further fine-tune the model, we could change the number of hidden units, values of the regularization parameters, learning rate, values of the decrease constant, or the adaptive learning using the techniques that we discussed in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning* (this is left as an exercise for the reader).
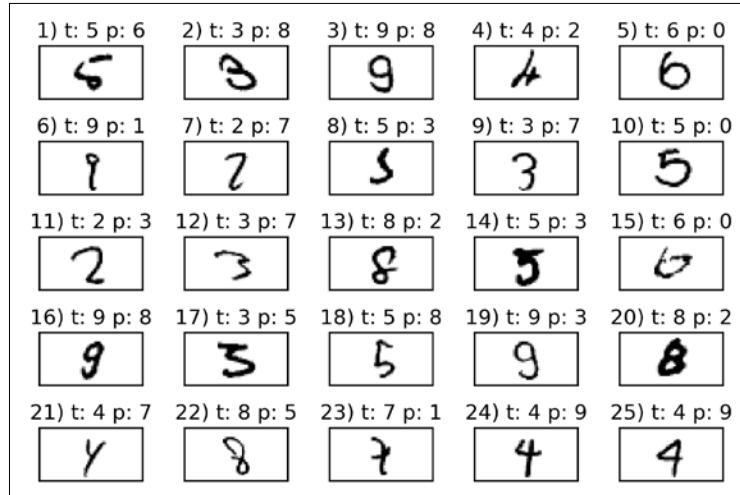
Now, let's take a look at some of the images that our MLP struggles with:

```
>>> miscl_img = X_test[y_test != y_test_pred][:25]
>>> correct_lab = y_test[y_test != y_test_pred][:25]
>>> miscl_lab= y_test_pred[y_test != y_test_pred][:25]

>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = miscl_img[i].reshape(28, 28)
...     ax[i].imshow(img,
...                  cmap='Greys',
...                  interpolation='nearest')
...     ax[i].set_title('%d) t: %d p: %d'
...                     % (i+1, correct_lab[i], miscl_lab[i]))
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

We should now see a 5×5 subplot matrix where the first number in the subtitles indicates the plot index, the second number indicates the true class label (t), and the third number stands for the predicted class label (p).

As we can see in the preceding figure, some of those images are even challenging for us humans to classify correctly. For example, we can see that the digit 9 is classified as a 3 or 8 if the lower part of the digit has a hook-like curvature (subplots 3, 16, and 17).

Training an artificial neural network

Now that we have seen a neural network in action and have gained a basic understanding of how it works by looking over the code, let's dig a little bit deeper into some of the concepts, such as the logistic cost function and the backpropagation algorithm that we implemented to learn the weights.

Computing the logistic cost function

The logistic cost function that we implemented as the `_get_cost` method is actually pretty simple to follow since it is the same cost function that we described in the logistic regression section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*.

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)$$

Here, $a^{(i)}$ is the sigmoid activation of the ith unit in one of the layers which we compute in the forward propagation step:

$$a^{(i)} = \phi\left(z^{(i)}\right)$$

Now, let's add a **regularization** term, which allows us to reduce the degree of overfitting. As you will recall from earlier chapters, the L2 and L1 regularization terms are defined as follows (remember that we don't regularize the bias units):

$$L2 = \lambda \left\|\boldsymbol{w}\right\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2 \text{ and } L1 = \lambda \left\|\boldsymbol{w}\right\|_1^1 = \lambda \sum_{j=1}^{m} \left|w_j\right|$$

Although our MLP implementation supports both L1 and L2 regularization, we will now only focus on the L2 regularization term for simplicity. However, the same concepts apply to the L1 regularization term. By adding the L2 regularization term to our logistic cost function, we obtain the following equation:

$$J(w) = -\left[\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)\right] + \frac{\lambda}{2} \|w\|_2^2$$

Since we implemented an MLP for multi-class classification, this returns an output vector of t elements, which we need to compare with the $t \times 1$ dimensional target vector in the one-hot encoding representation. For example, the activation of the third layer and the target class (here: class 2) for a particular sample may look like this:

$$a^{(3)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Thus, we need to generalize the logistic cost function to all activation units j in our network. So our cost function (without the regularization term) becomes:

$$J(w) = -\sum_{i=1}^{n} \sum_{j=1}^{t} y_j^{(i)} \log\left(a_j^{(i)}\right) + \left(1 - y_j^{(i)}\right) \log\left(1 - a_j^{(i)}\right)$$

Here, the superscript i is the index of a particular sample in our training set.

The following generalized regularization term may look a little bit complicated at first, but here we are just calculating the sum of all weights of a layer l (without the bias term) that we added to the first column:

$$J(w) = -\left[\sum_{i=1}^{n} \sum_{j=1}^{m} y_j^{(i)} \log\left(\phi\left(z_j^{(i)}\right)\right) + \left(1 - y_j^{(i)}\right) \log\left(1 - \phi\left(z_j^{(i)}\right)\right)\right]$$
$$+ \frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{u_l} \sum_{j=1}^{u_{l+1}} \left(w_{j,i}^{(l)}\right)^2$$

The following expression represents the L2-penalty term:
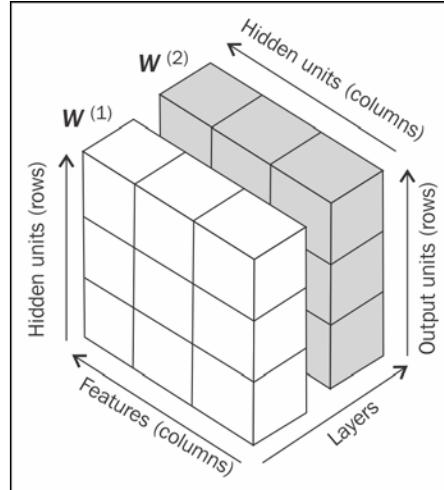
$$\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$$

Remember that our goal is to minimize the cost function $J(w)$. Thus, we need to calculate the partial derivative of matrix W with respect to each weight for every layer in the network:

$$\frac{\partial}{\partial w_{j,i}^{(l)}}J(W)$$

In the next section, we will talk about the backpropagation algorithm, which allows us to calculate these partial derivatives to minimize the cost function.

Note that W consists of multiple matrices. In a multi-layer perceptron with one hidden layer, we have the weight matrix $W^{(1)}$, which connects the input to the hidden layer, and $W^{(2)}$, which connects the hidden layer to the output layer. An intuitive visualization of the matrix W is provided in the following figure:

In this simplified figure, it may seem that both $W^{(1)}$ and $W^{(2)}$ have the same number of rows and columns, which is typically not the case unless we initialize an MLP with the same number of hidden units, output units, and input features.

If this may sound confusing, stay tuned for the next section where we will discuss the dimensionality of $W^{(1)}$ and $W^{(2)}$ in more detail in the context of the backpropagation algorithm.

Training neural networks via backpropagation

In this section, we will go through the math of backpropagation to understand how you can learn the weights in a neural network very efficiently. Depending on how comfortable you are with mathematical representations, the following equations may seem relatively complicated at first. Many people prefer a bottom-up approach and like to go over the equations step by step to develop an intuition for algorithms. However, if you prefer a top-down approach and want to learn about backpropagation without all the mathematical notations, I recommend you to read the next section *Developing your intuition for backpropagation* first and revisit this section later.

In the previous section, we saw how to calculate the cost as the difference between the activation of the last layer and the target class label. Now, we will see how the backpropagation algorithm works to update the weights in our MLP model, which we implemented in the _get_gradient method. As we recall from the beginning of this chapter, we first need to apply forward propagation in order to obtain the activation of the output layer, which we formulated as follows:
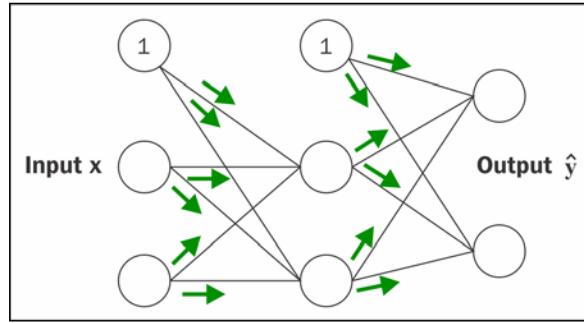
$$\boldsymbol{Z}^{(2)} = \boldsymbol{W}^{(1)} \left[\boldsymbol{A}^{(1)} \right]^{T} \left(\text{net input of the hidden layer} \right)$$

$$\boldsymbol{A}^{(2)} = \phi\left(\boldsymbol{Z}^{(2)} \right) \left(\text{activation of the hidden layer} \right)$$

$$\boldsymbol{Z}^{(3)} = \boldsymbol{W}^{(2)} \boldsymbol{A}^{(2)} \left(\text{net input of the output layer} \right)$$

$$\boldsymbol{A}^{(3)} = \phi\left(\boldsymbol{Z}^{(3)} \right) \left(\text{activation of the output layer} \right)$$

Concisely, we just forward propagate the input features through the connection in the network as shown here:

In backpropagation, we propagate the error from right to left. We start by calculating the error vector of the output layer:

$$\delta^{(3)} = a^{(3)} - y$$

Here, y is the vector of the true class labels.

Next, we calculate the error term of the hidden layer:

$$\boldsymbol{\delta}^{(2)} = \left(\boldsymbol{W}^{(2)}\right)^{T} \boldsymbol{\delta}^{(3)} * \frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}}$$

Here, $\frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}}$ is simply the derivative of the sigmoid activation function, which we implemented as `_sigmoid_gradient`:

$$\frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}} = \left(a^{(2)} * \left(1 - a^{(2)}\right)\right)$$

Note that the asterisk symbol $(*)$ means element-wise multiplication in this context.

Although, it is not important to follow the next equations, you may be curious as to how I obtained the derivative of the activation function. I summarized the derivation step by step here:

$$\phi'(z) = \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right)$$

$$= \frac{e^{-z}}{\left(1+e^{-z}\right)^2}$$

$$= \frac{1+e^{-z}}{\left(1+e^{-z}\right)^2} - \left(\frac{1}{1+e^{-z}}\right)^2$$

$$= \frac{1}{\left(1+e^{-z}\right)} - \left(\frac{1}{1+e^{-z}}\right)^2$$

$$= \phi(z) - \left(\phi(z)\right)^2$$

$$= \phi(z)\left(1-\phi(z)\right)$$

$$= a(1-a)$$

To better understand how we compute the $\delta^{(3)}$ term, let's walk through it in more detail. In the preceding equation, we multiplied the transpose $\left(W^{(2)}\right)^T$ of the $t \times h$ dimensional matrix $W^{(2)}$; t is the number of output class labels and h is the number of hidden units. Now, $\left(W^{(2)}\right)^T$ becomes an $h \times t$ dimensional matrix with $\delta^{(3)}$, which is a $t \times 1$ dimensional vector. We then performed a pair-wise multiplication between $\left(W^{(2)}\right)^T \delta^{(3)}$ and $\left(a^{(2)} * \left(1-a^{(2)}\right)\right)$, which is also a $t \times 1$ dimensional vector. Eventually, after obtaining the δ terms, we can now write the derivation of the cost function as follows:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$$

Next, we need to accumulate the partial derivative of every j th node in layer l and the ith error of the node in layer $l+1$:

$$\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

Remember that we need to compute $\Delta_{i,j}^{(l)}$ for every sample in the training set. Thus, it is easier to implement it as a vectorized version like in our preceding MLP code implementation:

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} \left(A^{(l)} \right)^T$$
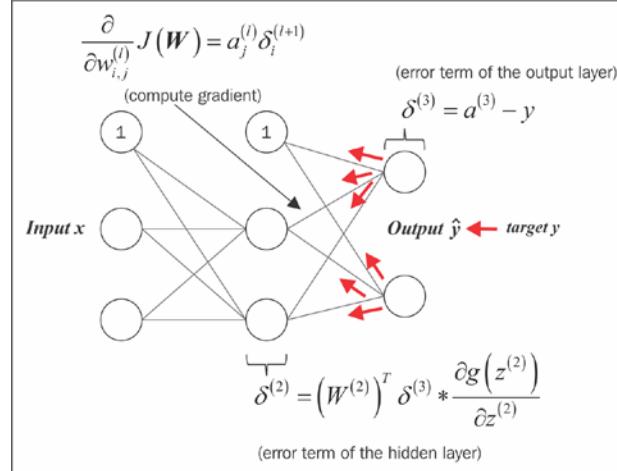
After we have accumulated the partial derivatives, we can add the regularization term as follows:

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \;\left(\text{except for the bias term} \right)$$

Lastly, after we have computed the gradients, we can now update the weights by taking an opposite step towards the gradient:

$$W^{(l)} := W^{(l)} - \eta \Delta^{(l)}$$

To bring everything together, let's summarize backpropagation in the following figure:

Developing your intuition for backpropagation

Although backpropagation was rediscovered and popularized almost 30 years ago, it still remains one of the most widely used algorithms to train artificial neural networks very efficiently. In this section, we'll see a more intuitive summary and the bigger picture of how this fascinating algorithm works.

In essence, backpropagation is just a very computationally efficient approach to compute the derivatives of a complex cost function. Our goal is to use those derivatives to learn the weight coefficients for parameterizing a multi-layer artificial neural network. The challenge in the parameterization of neural networks is that we are typically dealing with a very large number of weight coefficients in a high-dimensional feature space. In contrast to other cost functions that we have seen in previous chapters, the error surface of a neural network cost function is not convex or smooth. There are many bumps in this high-dimensional cost surface (local minima) that we have to overcome in order to find the global minimum of the cost function.

You may recall the concept of the chain rule from your introductory calculus classes. The chain rule is an approach to deriving a complex, nested function, for example, $f(g(x)) = y$ that is broken down into basic components:

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$$

In the context of computer algebra, a set of techniques has been developed to solve such problems very efficiently, which is also known as *automatic differentiation*. If you are interested in learning more about automatic differentiation in machine learning applications, I recommend you to refer to the following resource: A. G. Baydin and B. A. Pearlmutter. *Automatic Differentiation of Algorithms for Machine Learning*. arXiv preprint arXiv:1404.7456, 2014, which is freely available on arXiv at `http://arxiv.org/pdf/1404.7456.pdf`.

Automatic differentiation comes with two modes, the *forward* and the *reverse* mode, respectively. Backpropagation is simply just a special case of the reverse-mode automatic differentiation. The key point is that applying the chain rule in the forward mode can be quite expensive since we would have to multiply large matrices for each layer (Jacobians) that we eventually multiply by a vector to obtain the output. The trick of the reverse mode is that we start from right to left: we multiply a matrix by a vector, which yields another vector that is multiplied by the next matrix and so on. Matrix-vector multiplication is computationally much cheaper than matrix-matrix multiplication, which is why backpropagation is one of the most popular algorithms used in neural network training.

Debugging neural networks with gradient checking

Implementations of artificial neural networks can be quite complex, and it is always a good idea to *manually* check that we have implemented backpropagation correctly. In this section, we will talk about a simple procedure called *gradient checking*, which is essentially a comparison between our analytical gradients in the network and numerical gradients. Gradient checking is not specific to feedforward neural networks but can be applied to any other neural network architecture that uses gradient-based optimization. Even if you are planning to implement more trivial algorithms using gradient-based optimization, such as linear regression, logistic regression, and support vector machines, it is generally not a bad idea to check if the gradients are computed correctly.
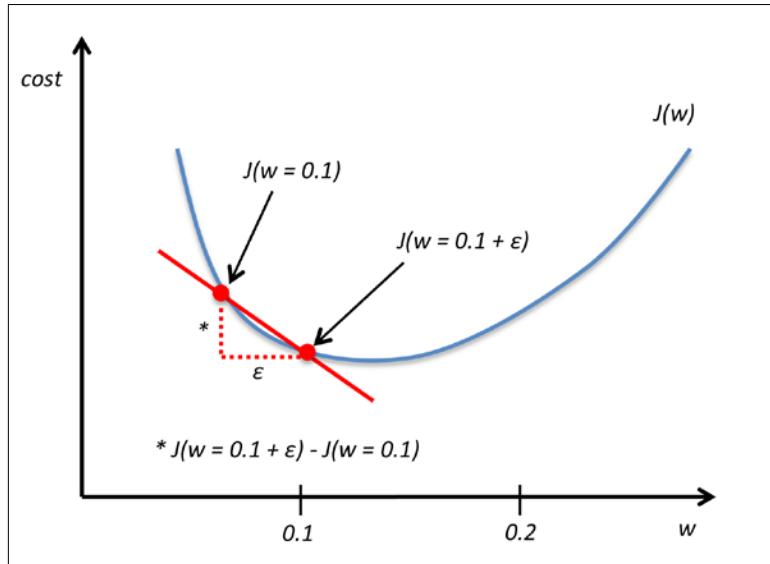
In the previous sections, we defined a cost function $J(W)$ where W is the matrix of the weight coefficients of an artificial network. Note that $J(W)$ is—roughly speaking—a "stacked" matrix consisting of the matrices $W^{(1)}$ and $W^{(2)}$ in a multi-layer perceptron with one hidden unit. We defined $W^{(1)}$ as the $h\times[m+1]$-dimensional matrix that connects the input layer to the hidden layer, where h is the number of hidden units and m is the number of features (input units). The matrix $W^{(2)}$ that connects the hidden layer to the output layer has the dimensions $t\times h$, where t is the number of output units. We then calculated the derivative of the cost function for a weight $w_{i,j}^{(l)}$ as follows:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W)$$

Remember that we are updating the weights by taking an opposite step towards the direction of the gradient. In gradient checking, we compare this analytical solution to a numerically approximated gradient:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) \approx \frac{J\left(w_{i,j}^{(l)} + \varepsilon\right) - J\left(w_{i,j}^{(l)}\right)}{\varepsilon}$$

Here, ε is typically a very small number, for example 1e-5 (note that 1e-5 is just a more convenient notation for 0.00001). Intuitively, we can think of this finite difference approximation as the slope of the secant line connecting the points of the cost function for the two weights w and $w + \varepsilon$ (both are scalar values), as shown in the following figure. We are omitting the superscripts and subscripts for simplicity.

An even better approach that yields a more accurate approximation of the gradient is to compute the symmetric (or centered) difference quotient given by the two-point formula:

$$\frac{J\left(w_{i,j}^{(l)} + \varepsilon\right) - J\left(w_{i,j}^{(l)} - \varepsilon\right)}{2\varepsilon}$$

Typically, the approximated difference between the numerical gradient J'_n and analytical gradient J'_a is then calculated as the L2 vector norm. For practical reasons, we unroll the computed gradient matrices into flat vectors so that we can calculate the error (the difference between the gradient vectors) more conveniently:

$$error = \left\| J'_n - J'_a \right\|_2$$

One problem is that the error is not scale invariant (small errors are more significant if the weight vector norms are small too). Thus, it is recommended to calculate a normalized difference:

$$relative\,error = \frac{\left\| J'_n - J'_a \right\|_2}{\left\| J'_n \right\|_2 + \left\| J'_a \right\|_2}$$

Now, we want the relative error between the numerical gradient and the analytical gradient to be as small as possible. Before we implement gradient checking, we need to discuss one more detail: what is the acceptable error threshold to pass the gradient check? The relative error threshold for passing the gradient check depends on the complexity of the network architecture. As a rule of thumb, the more hidden layers we add, the larger the difference between the numerical and analytical gradient can become if backpropagation is implemented correctly. Since we have implemented a relatively simple neural network architecture in this chapter, we want to be rather strict about the threshold and define the following rules:

- Relative error <= 1e-7 means everything is okay!
- Relative error <= 1e-4 means the condition is problematic, and we should look into it.
- Relative error > 1e-4 means there is probably something wrong in our code.

Now that we have established these ground rules, let's implement gradient checking. To do so, we can simply take the `NeuralNetMLP` class that we implemented previously and add the following method to the class body:

```
def _gradient_checking(self, X, y_enc, w1,
                       w2, epsilon, grad1, grad2):
    """ Apply gradient checking (for debugging only)

    Returns
    ---------
```

```
    relative_error : float
      Relative error between the numerically
      approximated gradients and the backpropagated gradients.

    """
    num_grad1 = np.zeros(np.shape(w1))
    epsilon_ary1 = np.zeros(np.shape(w1))
    for i in range(w1.shape[0]):
        for j in range(w1.shape[1]):
            epsilon_ary1[i, j] = epsilon
            a1, z2, a2, z3, a3 = self._feedforward(
                                        X,
                                        w1 - epsilon_ary1,
                                        w2)
            cost1 = self._get_cost(y_enc,
                                    a3,
                                    w1-epsilon_ary1,
                                    w2)
            a1, z2, a2, z3, a3 = self._feedforward(
                                        X,
                                        w1 + epsilon_ary1,
                                        w2)
            cost2 = self._get_cost(y_enc,
                                    a3,
                                    w1 + epsilon_ary1,
                                    w2)
            num_grad1[i, j] = (cost2 - cost1) / (2 * epsilon)
            epsilon_ary1[i, j] = 0

    num_grad2 = np.zeros(np.shape(w2))
    epsilon_ary2 = np.zeros(np.shape(w2))
    for i in range(w2.shape[0]):
        for j in range(w2.shape[1]):
            epsilon_ary2[i, j] = epsilon
            a1, z2, a2, z3, a3 = self._feedforward(
                                        X,
                                        w1,
                                        w2 - epsilon_ary2)
            cost1 = self._get_cost(y_enc,
                                    a3,
                                    w1,
                                    w2 - epsilon_ary2)
            a1, z2, a2, z3, a3 = self._feedforward(
```

```
                                            X,
                                            w1,
                                            w2 + epsilon_ary2)
                cost2 = self._get_cost(y_enc,
                                        a3,
                                        w1,
                                        w2 + epsilon_ary2)
                num_grad2[i, j] = (cost2 - cost1) / (2 * epsilon)
                epsilon_ary2[i, j] = 0

        num_grad = np.hstack((num_grad1.flatten(),
                              num_grad2.flatten()))
        grad = np.hstack((grad1.flatten(), grad2.flatten()))
        norm1 = np.linalg.norm(num_grad - grad)
        norm2 = np.linalg.norm(num_grad)
        norm3 = np.linalg.norm(grad)
        relative_error = norm1 / (norm2 + norm3)
        return relative_error
```

The `_gradient_checking` code seems rather simple. However, my personal
recommendation is to keep it as simple as possible. Our goal is to double-check
the gradient computation, so we want to make sure that we do not introduce any
additional mistakes in gradient checking by writing efficient but complex code.
Next, we only need to make a small modification to the `fit` method. In the following
code, I omitted the code at the beginning of the `fit` function for clarity, and the only
lines that we need to add to the method are implemented between the comments `##
start gradient checking` and `## end gradient checking`:

```
class MLPGradientCheck(object):
    [...]
    def fit(self, X, y, print_progress=False):
        [...]
                # compute gradient via backpropagation
                grad1, grad2 = self._get_gradient(
                                    a1=a1,
                                    a2=a2,
                                    a3=a3,
                                    z2=z2,
                                    y_enc=y_enc[:, idx],
                                    w1=self.w1,
                                    w2=self.w2)


                ## start gradient checking
```

```
                   grad_diff = self._gradient_checking(
                                   X=X[idx],
                                   y_enc=y_enc[:, idx],
                                   w1=self.w1,
                                   w2=self.w2,
                                   epsilon=1e-5,
                                   grad1=grad1,
                                   grad2=grad2)
               if grad_diff <= 1e-7:
                   print('Ok: %s' % grad_diff)
               elif grad_diff <= 1e-4:
                   print('Warning: %s' % grad_diff)
               else:
                   print('PROBLEM: %s' % grad_diff)

               ## end gradient checking

               # update weights; [alpha * delta_w_prev]
               # for momentum learning
               delta_w1 = self.eta * grad1
               delta_w2 = self.eta * grad2
               self.w1 -= (delta_w1 +\
                           (self.alpha * delta_w1_prev))
               self.w2 -= (delta_w2 +\
                           (self.alpha * delta_w2_prev))
               delta_w1_prev = delta_w1
               delta_w2_prev = delta_w2

       return self
```

Assuming that we named our modified multi-layer perceptron class
`MLPGradientCheck`, we can now initialize a new MLP with 10 hidden layers. Also,
we disable regularization, adaptive learning, and momentum learning. In addition,
we use regular gradient descent by setting `minibatches` to 1. The code is as follows:

```
>>> nn_check = MLPGradientCheck(n_output=10,
                                n_features=X_train.shape[1],
                                n_hidden=10,
                                l2=0.0,
                                l1=0.0,
                                epochs=10,
                                eta=0.001,
                                alpha=0.0,
                                decrease_const=0.0,
                                minibatches=1,
                                random_state=1)
```

One downside of gradient checking is that it is computationally very, very expensive. Training a neural network with gradient checking enabled is so slow that we really only want to use it for debugging purposes. For this reason, it is not uncommon to run gradient checking only on a handful of training samples (here, we choose 5). The code is as follows:
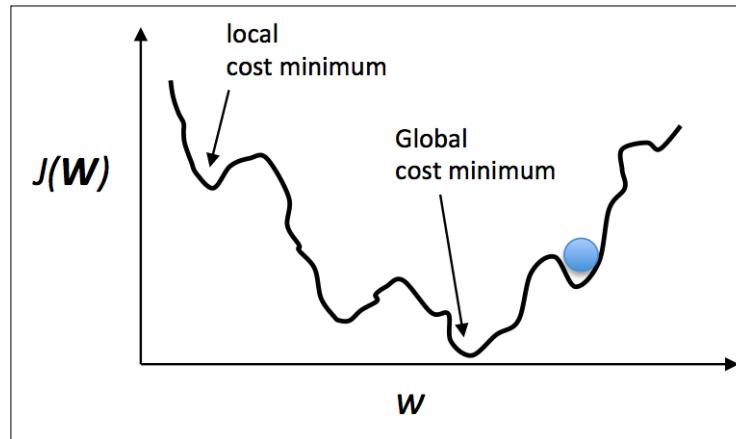
```
>>> nn_check.fit(X_train[:5], y_train[:5], print_progress=False)
Ok: 2.56712936241e-10
Ok: 2.94603251069e-10
Ok: 2.37615620231e-10
Ok: 2.43469423226e-10
Ok: 3.37872073158e-10
Ok: 3.63466384861e-10
Ok: 2.22472120785e-10
Ok: 2.33163708438e-10
Ok: 3.44653686551e-10
Ok: 2.17161707211e-10
```

As we can see from the code output, our multi-layer perceptron passes this test with excellent results.

Convergence in neural networks

You might be wondering why we did not use regular gradient descent but mini-batch learning to train our neural network for the handwritten digit classification. You may recall our discussion on stochastic gradient descent that we used to implement online learning. In online learning, we compute the gradient based on a single training example ($k = 1$) at a time to perform the weight update. Although this is a stochastic approach, it often leads to very accurate solutions with a much faster convergence than regular gradient descent. Mini-batch learning is a special form of stochastic gradient descent where we compute the gradient based on a subset k of the n training samples with $1 < k < n$. Mini-batch learning has the advantage over online learning that we can make use of our vectorized implementations to improve computational efficiency. However, we can update the weights much faster than in regular gradient descent. Intuitively, you can think of mini-batch learning as predicting the vote turnout of a presidential election from a poll by asking only a representative subset of the population rather than asking the entire population.

In addition, we added more tuning parameters such as the decrease constant and a parameter for an adaptive learning rate. The reason is that neural networks are much harder to train than simpler algorithms such as Adaline, logistic regression, or support vector machines. In multi-layer neural networks, we typically have hundreds, thousands, or even billions of weights that we need to optimize. Unfortunately, the output function has a rough surface and the optimization algorithm can easily become trapped in local minima, as shown in the following figure:

Note that this representation is extremely simplified since our neural network has many dimensions; it makes it impossible to visualize the actual cost surface for the human eye. Here, we only show the cost surface for a single weight on the x axis. However, the main message is that we do not want our algorithm to get trapped in local minima. By increasing the learning rate, we can more readily escape such local minima. On the other hand, we also increase the chance of overshooting the global optimum if the learning rate is too large. Since we initialize the weights randomly, we start with a solution to the optimization problem that is typically hopelessly wrong. A decrease constant, which we defined earlier, can help us to climb down the cost surface faster in the beginning and the adaptive learning rate allows us to better anneal to the global minimum.

Other neural network architectures

In this chapter, we discussed one of the most popular feedforward neural network representations, the multi-layer perceptron. Neural networks are currently one of the most active research topics in the machine learning field, and there are many other neural network architectures that are well beyond the scope of this book. If you are interested in learning more about neural networks and algorithms for deep learning, I recommend reading the introduction and overview; Y. Bengio. *Learning Deep Architectures for AI*. Foundations and Trends in Machine Learning, 2(1):1–127, 2009. Yoshua Bengio's book is currently freely available at `http://www.iro.umontreal.ca/~bengioy/papers/ftml_book.pdf`.
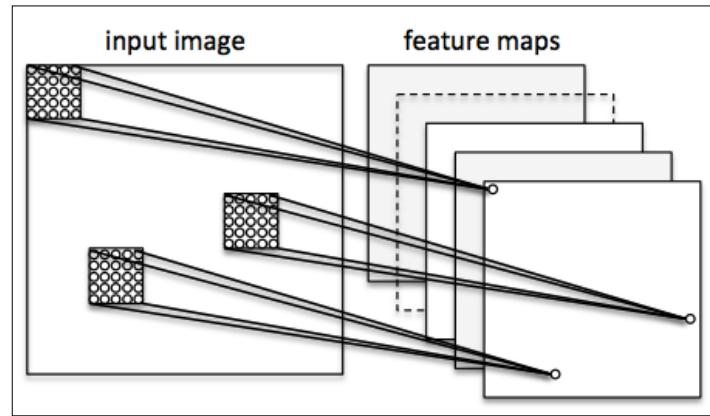
Although neural networks really are a topic for another book, let's take at least a brief look at two other popular architectures, **convolutional neural networks** and **recurrent neural networks**.

Convolutional Neural Networks

Convolutional Neural Networks (**CNNs** or **ConvNets**) gained popularity in computer vision due to their extraordinary good performance on image classification tasks. As of today, CNNs are one of the most popular neural network architectures in deep learning. The key idea behind convolutional neural networks is to build many layers of **feature detectors** to take the spatial arrangement of pixels in an input image into account. Note that there exist many different variants of CNNs. In this section, we will discuss only the general idea behind this architecture. If you are interested in learning more about CNNs, I recommend you to take a look at the publications of Yann LeCun (`http://yann.lecun.com`), who is one of the co-inventors of CNNs. In particular, I can recommend the following literature for getting started with CNNs:
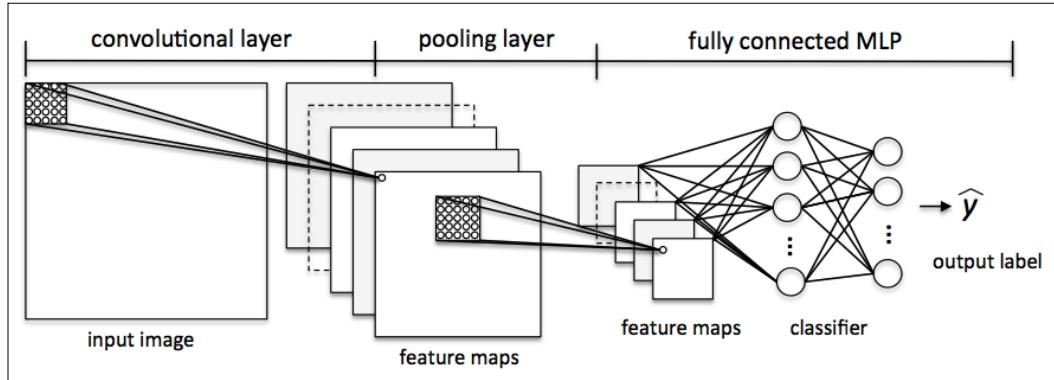
- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. *Gradient-based Learning Applied to Document Recognition*. Proceedings of the IEEE, 86(11):2278–2324, 1998.

- P. Y. Simard, D. Steinkraus, and J. C. Platt. *Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis*. IEEE, 2003, p.958.

As you will recall from our multi-layer perceptron implementation, we unrolled the images into feature vectors and these inputs were fully connected to the hidden layer—spatial information was not encoded in this network architecture. In CNNs, we use **receptive fields** to connect the input layer to a feature map. These receptive fields can be understood as overlapping windows that we slide over the pixels of an input image to create a feature map. The stride lengths of the window sliding as well as the window size are additional hyperparameters of the model that we need to define *a priori*. The process of creating the **feature map** is also called **convolution**. An example of such a **convolutional layer**, the layer that connects the input pixels to each unit in the feature map, is shown in the following figure:
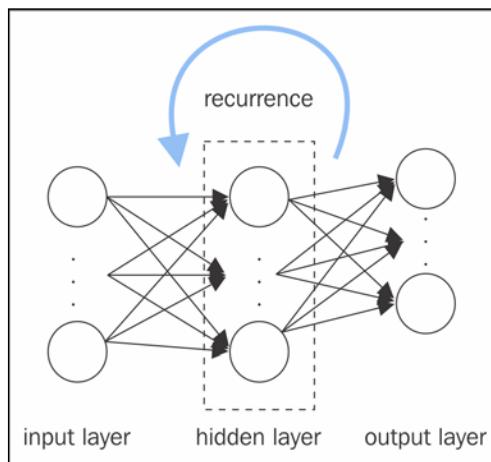
It is important to note that the feature detectors are replicates, which means that the receptive fields that map the features to the units in the next layer share the same weights. Here, the key idea is that if a feature detector is useful in one part of the image, it might be useful in another part as well. The nice side effect of this approach is that it greatly reduces the number of parameters that need to be learned. Since we allow different patches of the image to be represented in different ways, CNNs are particularly good at recognizing objects of different sizes and different positions in an image. We do not need to worry so much about rescaling and centering the images as it has been done in MNIST.

In CNNs, a convolutional layer is followed by a **pooling layer** (sometimes also called **sub-sampling**). In pooling, we summarize neighboring feature detectors to reduce the number of features for the next layer. Pooling can be understood as a simple method of feature extraction where we take the average or maximum value of a patch of neighboring features and pass it on to the next layer. To create a deep convolutional neural network, we stack multiple layers—alternating between convolutional and pooling layers—before we connect it to a multi-layer perceptron for classification. This is shown in the following figure:

Recurrent Neural Networks

Recurrent Neural Networks (**RNNs**) can be thought of as feedforward neural networks with feedback loops or backpropagation through time. In RNNs, the neurons only fire for a limited amount of time before they are (temporarily) deactivated. In turn, these neurons activate other neurons that fire at a later point in time. Basically, we can think of recurrent neural networks as MLPs with an additional time variable. The time component and dynamic structure allows the network to use not only the current inputs but also the inputs that it encountered earlier.

Although RNNs achieved remarkable results in speech recognition, language translation, and connected handwriting recognition, these network architectures are typically much harder to train. This is because we cannot simply backpropagate the error layer by layer; we have to consider the additional time component, which amplifies the vanishing and exploding gradient problem. In 1997, Juergen Schmidhuber and his co-workers introduced the so-called long short-term memory units to overcome this problem: **Long Short Term Memory** (**LSTM**) units; S. Hochreiter and J. Schmidhuber. *Long Short-term Memory*. Neural Computation, 9(8):1735–1780, 1997.

However, we should note that there are many different variants of RNNs, and a detailed discussion is beyond the scope of this book.

A few last words about neural network implementation

You might be wondering why we went through all of this theory just to implement a simple multi-layer artificial network that can classify handwritten digits instead of using an open source Python machine learning library. One reason is that at the time of writing this book, scikit-learn does not have an MLP implementation. More importantly, we (machine learning practitioners) should have at least a basic understanding of the algorithms that we are using in order to apply machine learning techniques appropriately and successfully.

Now that we know how feedforward neural networks work, we are ready to explore more sophisticated Python libraries built on top of NumPy such as Theano (`http://deeplearning.net/software/theano/`), which allows us to construct neural networks more efficiently. We will see this in *Chapter 13*, *Parallelizing Neural Network Training with Theano*. Over the last couple of years, Theano has gained a lot of popularity among machine learning researchers, who use it to construct deep neural networks because of its ability to optimize mathematical expressions for computations on multi-dimensional arrays utilizing **Graphical Processing Units** (**GPUs**).

A great collection of Theano tutorials can be found at `http://deeplearning.net/ software/theano/tutorial/index.html#tutorial`.

There are also a number of interesting libraries that are being actively developed to train neural networks in Theano, which you should keep on your radar:

- **Pylearn2** (`http://deeplearning.net/software/pylearn2/`)
- **Lasagne** (`https://lasagne.readthedocs.org/en/latest/`)
- **Keras** (`http://keras.io`)

Summary

In this chapter, you have learned about the most important concepts behind multi-layer artificial neural networks, which are currently the hottest topic in machine learning research. In *Chapter 2*, *Training Machine Learning Algorithms for Classification*, we started our journey with simple single-layer neural network structures and now we have connected multiple neurons to a powerful neural network architecture to solve complex problems such as handwritten digit recognition. We demystified the popular backpropagation algorithm, which is one of the building blocks of many neural network models that are used in deep learning. After learning about the backpropagation algorithm, we were able to update the weights of such a complex neural network. We also added useful modifications such as mini-batch learning and an adaptive learning rate that allows us to train a neural network more efficiently.

13

Parallelizing Neural Network Training with Theano

In the previous chapter, we went over a lot of mathematical concepts to understand how feedforward artificial neural networks and multilayer perceptrons in particular work. First and foremost, having a good understanding of the mathematical underpinnings of machine learning algorithms is very important, since it helps us to use those powerful algorithms most effectively and *correctly*. Throughout the previous chapters, you dedicated a lot of time to learning the best practices of machine learning, and you even practiced implementing algorithms yourself from scratch. In this chapter, you can lean back a little bit and rest on your laurels, I want you to enjoy this exciting journey through one of the most powerful libraries that is used by machine learning researchers to experiment with deep neural networks and train them very efficiently. Most of modern machine learning research utilizes computers with powerful **Graphics Processing Units** (**GPUs**). If you are interested in diving into deep learning, which is currently the hottest topic in machine learning research, this chapter is definitely for you. However, do not worry if you do not have access to GPUs; in this chapter, the use of GPUs will be optional, not required.

Before we get started, let me give you a brief overview of the topics that we will cover in this chapter:

- Writing optimized machine learning code with Theano
- Choosing activation functions for artificial neural networks
- Using the Keras deep learning library for fast and easy experimentation

Building, compiling, and running expressions with Theano

In this section, we will explore the powerful Theano tool, which has been designed to train machine learning models most effectively using Python. The Theano development started back in 2008 in the **LISA** lab (short for **Laboratoire d'Informatique des Systèmes Adaptatifs** (`http://lisa.iro.umontreal.ca`)) lead by Yoshua Bengio.

Before we discuss what Theano really is and what it can do for us to speed up our machine learning tasks, let's discuss some of the challenges when we are running expensive calculations on our hardware. Luckily, the performance of computer processors keeps on improving constantly over the years, which allows us to train more powerful and complex learning systems to improve the predictive performance of our machine learning models. Even the cheapest desktop computer hardware that is available nowadays comes with processing units that have multiple cores. In the previous chapters, we saw that many functions in scikit-learn allow us to spread the computations over multiple processing units. However, by default, Python is limited to execution on one core, due to the **Global Interpreter Lock** (**GIL**). However, although we take advantage of its `multiprocessing` library to distribute computations over multiple cores, we have to consider that even advanced desktop hardware rarely comes with more than 8 or 16 such cores.

If we think back of the previous chapter where we implemented a very simple multilayer perceptron with only one hidden layer consisting of 50 units, we already had to optimize approximately 1000 weights to learn a model for a very simple image classification task. The images in MNIST are rather small (28 x 28 pixels), and we can only imagine the explosion in the number of parameters if we want to add additional hidden layers or work with images that have higher pixel densities. Such a task would quickly become unfeasible for a single processing unit. Now, the question is how can we tackle such problems more effectively? The obvious solution to this problem is to use GPUs. GPUs are real power horses. You can think of a graphics card as a small computer cluster inside your machine. Another advantage is that modern GPUs are relatively cheap compared to the state-of-the-art CPUs, as we can see in the following overview:

Specifications	Intel® Core™ i7-5960X Processor Extreme Edition	NVIDIA GeForce® GTX™ 980 Ti
Base Clock Frequency	3.0 GHz	1.0 GHz
Cores	8	2816
Memory Bandwidth	68 GB/s	336.5 GB/s
Floating-Point Calculations	354 GFLOPS	5632 GFLOPS
Cost	$1000.00	$700.00

Sources for this can be found on the following websites:

- `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980-ti/specifications`
- `http://ark.intel.com/products/82930/Intel-Core-i7-5960X-Processor-Extreme-Edition-20M-Cache-up-to-3_50-GHz`

(date: August 20, 2015)

At 70 percent of the price of a modern CPU, we can get a GPU that has 450 times more cores, and is capable of around 15 times more floating-point calculations per second. So, what is holding us back from utilizing GPUs for our machine learning tasks? The challenge is that writing code to target GPUs is not as trivial as executing Python code in our interpreter. There are special packages such as CUDA and OpenCL that allow us to target the GPU. However, writing code in CUDA or OpenCL is probably not the most convenient environment for implementing and running machine learning algorithms. The good news is that this is what Theano was developed for!

What is Theano?

What exactly is Theano—a programming language, a compiler, or a Python library? It turns out that it fits all these descriptions. Theano has been developed to implement, compile, and evaluate mathematical expressions very efficiently with a strong focus on multidimensional arrays (tensors). It comes with an option to run code on CPU(s). However, its real power comes from utilizing GPUs to take advantage of the large memory bandwidths and great capabilities for floating point math. Using Theano, we can easily run code in parallel over shared memory as well. In 2010, the developers of Theano reported an 1.8x faster performance than NumPy when the code was run on the CPU, and if Theano targeted the GPU, it was even 11x faster than NumPy (J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. *Theano: A CPU and GPU Math Compiler in Python*. In Proc. 9th Python in Science Conf, pages 1–7, 2010.). Now, keep in mind that this benchmark is from 2010, and Theano has improved significantly over the years, and so have the capabilities of modern graphics cards.

So, how does Theano relate to NumPy? Theano is built on top of NumPy and it has a very similar syntax, which makes the usage very convenient for people who are already familiar with the latter. To be fair, Theano is not just "NumPy on steroids" as many people would describe it, but it also shares some similarities with SymPy (`http://www.sympy.org`), a Python package for symbolic computations (or symbolic algebra). As we saw in previous chapters, in NumPy, we describe what our variables are, and how we want to combine them; then, the code is executed line by line. In Theano, however, we write down the problem first and the description of how we want to analyze it. Then, Theano optimizes and compiles code for us using C/C++, or CUDA/OpenCL if we want to run it on the GPU. In order to generate the optimized code for us, Theano needs to know the scope of our problem; think of it as a tree of operations (or a graph of symbolic expressions). Note that Theano is still under active development, and many new features are added and improvements are made on a regular basis. In this chapter, we will explore the basic concepts behind Theano and learn how to use it for machine learning tasks. Since Theano is a large library with many advanced features, it would be impossible to cover all of them in this book. However, I will provide useful links to the excellent online documentation (`http://deeplearning.net/software/theano/`) if you want to learn more about this library.

First steps with Theano

In this section, we will take our first steps with Theano. Depending on how your system is set up, you typically can just use the pip installer and install Theano from PyPI by executing the following from your command-line terminal:

```
pip install Theano
```

If you should experience problems with the installation procedure, I recommend you to read more about system and platform-specific recommendations that are provided at http://deeplearning.net/software/theano/install.html. Note that all the code in this chapter can be run on your CPU; using the GPU is entirely optional but recommended if you fully want to enjoy the benefits of Theano. If you have a graphics card that supports either CUDA or OpenCL, please refer to the up-to-date tutorial at http://deeplearning.net/software/theano/tutorial/using_gpu.html#using-gpu to set it up appropriately.

At its core, Theano is built around so-called tensors to evaluate symbolic mathematical expressions. Tensors can be understood as a generalization of scalars, vectors, matrices, and so on. More concretely, a scalar can be defined as a rank-0 tensor, a vector as a rank-1 tensor, a matrix as rank-2 tensor, and matrices stacked in a third dimension as rank-3 tensors. As a warm-up exercise, we will start with the use of simple scalars from the Theano `tensor` module to compute a net input z of a sample point x in a one dimensional dataset with weight w_1 and bias w_0:

$$z = x_1 \times w_1 + w_0$$

The code is as follows:

```
>>> import theano
>>> from theano import tensor as T

# initialize
>>> x1 = T.scalar()
>>> w1 = T.scalar()
>>> w0 = T.scalar()
>>> z1 = w1 * x1 + w0

# compile
>>> net_input = theano.function(inputs=[w1, x1, w0],
...                             outputs=z1)

# execute
>>> print('Net input: %.2f' % net_input(2.0, 1.0, 0.5))
Net input: 2.50
```

This was pretty straightforward, right? If we write code in Theano, we just have to follow three simple steps: define the *symbols* (`Variable` objects), compile the code, and execute it. In the initialization step, we defined three symbols, `x1`, `w1`, and `w0`, to compute `z1`. Then, we compiled a function `net_input` to compute the net input `z1`.

However, there is one particular detail that deserves special attention if we write Theano code: the type of our variables (`dtype`). Consider it as a blessing or burden, but in Theano we need to choose whether we want to use 64 or 32 bit integers or floats, which greatly affects the performance of the code. Let's discuss those variable types in more detail in the next section.

Configuring Theano

Nowadays, no matter whether we run Mac OS X, Linux, or Microsoft Windows, we mainly use software and applications using 64-bit memory addresses. However, if we want to accelerate the evaluation of mathematical expressions on GPUs, we still often rely on the older 32-bit memory addresses. Currently, this is the only supported computing architecture in Theano. In this section, we will see how to configure Theano appropriately. If you are interested in more details about the Theano configuration, please refer to the online documentation at http://deeplearning.net/software/theano/library/config.html.

When we are implementing machine learning algorithms, we are mostly working with floating point numbers. By default, both NumPy and Theano use the double-precision floating-point format (`float64`). However, it would be really useful to toggle back and forth `float64` (CPU), and `float32` (GPU) when we are developing Theano code for prototyping on CPU and execution on GPU. For example, to access the default settings for Theano's float variables, we can execute the following code in our Python interpreter:

```
>>> print(theano.config.floatX)
float64
```

If you have not modified any settings after the installation of Theano, the floating point default should be `float64`. However, we can simply change it to `float32` in our current Python session via the following code:

```
>>> theano.config.floatX = 'float32'
```

Note that although the current GPU utilization in Theano requires `float32` types, we can use both `float64` and `float32` on our CPUs. Thus, if you want to change the default settings globally, you can change the settings in your `THEANO_FLAGS` variable via the command-line (Bash) terminal:

```
export THEANO_FLAGS=floatX=float32
```

Alternatively, you can apply these settings only to a particular Python script, by running it as follows:

```
THEANO_FLAGS=floatX=float32 python your_script.py
```

So far, we discussed how to set the default floating-point types to get the best bang for the buck on our GPU using Theano. Next, let's discuss the options to toggle between CPU and GPU execution. If we execute the following code, we can check whether we are using CPU or GPU:

```
>>> print(theano.config.device)
cpu
```

My personal recommendation is to use `cpu` as default, which makes prototyping and code debugging easier. For example, you can run Theano code on your CPU by executing it as a script, as from your command-line terminal:

```
THEANO_FLAGS=device=cpu,floatX=float64 python your_script.py
```

However, once we have implemented the code and want to run it most efficiently utilizing our GPU hardware, we can then run it via the following code without making additional modifications to our original code:

```
THEANO_FLAGS=device=gpu,floatX=float32 python your_script.py
```

It may also be convenient to create a `.theanorc` file in your home directory to make these configurations permanent. For example, to always use `float32` and the GPU, you can create such a `.theanorc` file including these settings. The command is as follows:

```
echo -e "\n[global]\nfloatX=float32\ndevice=gpu\n" >> ~/.theanorc
```

If you are not operating on a MacOS X or Linux terminal, you can create a `.theanorc` file manually using your favorite text editor and add the following contents:

```
[global]
floatX=float32
device=gpu
```

Now that we know how to configure Theano appropriately with respect to our available hardware, we can discuss how to use more complex array structures in the next section.

Working with array structures

In this section, we will discuss how to use array structures in Theano using its `tensor` module. By executing the following code, we will create a simple 2 x 3 matrix, and calculate the column sums using Theano's optimized tensor expressions:

```
>>> import numpy as np

# initialize
# if you are running Theano on 64 bit mode,
# you need to use dmatrix instead of fmatrix
>>> x = T.fmatrix(name='x')
>>> x_sum = T.sum(x, axis=0)

# compile
>>> calc_sum = theano.function(inputs=[x], outputs=x_sum)

# execute (Python list)
>>> ary = [[1, 2, 3], [1, 2, 3]]
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2.  4.  6.]

# execute (NumPy array)
>>> ary = np.array([[1, 2, 3], [1, 2, 3]],
...                 dtype=theano.config.floatX)
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2.  4.  6.]
```

As we saw earlier, there are just three basic steps that we have to follow when we are using Theano: defining the variable, compiling the code, and executing it. The preceding example shows that Theano can work with both Python and NumPy types: `list` and `numpy.ndarray`.

> Note that we used the optional name argument (here, x) when we created the `fmatrix` TensorVariable, which can be helpful to debug our code or print the Theano graph. For example, if we'd print the `fmatrix` symbol x without giving it a `name`, the `print` function would return its TensorType:
>
> ```
> >>> print(x)
> <TensorType(float32, matrix)>
> ```
>
> However, if the `TensorVariable` was initialized with a `name` argument x as in our preceding example, it would be returned by the `print` function:
>
> ```
> >>> print(x)
> x
> ```
>
> The `TensorType` can be accessed via the `type` method:
>
> ```
> >>> print(x.type())
> <TensorType(float32, matrix)>
> ```

Theano also has a very smart memory management system that reuses memory to make it fast. More concretely, Theano spreads memory space across multiple devices, CPUs and GPUs; to track changes in the memory space, it aliases the respective buffers. Next, we will take a look at the `shared` variable, which allows us to spread large objects (arrays) and grants multiple functions read and write access, so that we can also perform updates on those objects after compilation. A detailed description of the memory handling in Theano is beyond the scope of this book. Thus, I encourage you to follow-up on the up-to-date information about Theano and memory management at `http://deeplearning.net/software/theano/tutorial/aliasing.html`.

```
# initialize
>>> x = T.fmatrix('x')
>>> w = theano.shared(np.asarray([[0.0, 0.0, 0.0]],
                              dtype=theano.config.floatX))
>>> z = x.dot(w.T)
>>> update = [[w, w + 1.0]]

# compile
>>> net_input = theano.function(inputs=[x],
...                             updates=update,
...                             outputs=z)

# execute
>>> data = np.array([[1, 2, 3]],
...                 dtype=theano.config.floatX)
>>> for i in range(5):
...     print('z%d:' % i, net_input(data))
z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

As you can see, sharing memory via Theano is really easy: In the preceding example, we defined an `update` variable where we declared that we want to update an array `w` by a value `1.0` after each iteration in the `for` loop. After we defined which object we want to update and how, we passed this information to the `update` parameter of the `theano.function` compiler.

Another neat trick in Theano is to use the `givens` variable to insert values into the graph before compiling it. Using this approach, we can reduce the number of transfers from RAM over CPUs to GPUs to speed up learning algorithms that use shared variables. If we use the `inputs` parameter in `theano.function`, data is transferred from the CPU to the GPU multiple times, for example, if we iterate over a dataset multiple times (epochs) during gradient descent. Using `givens`, we can keep the dataset on the GPU if it fits into its memory (for example, if we are learning with mini-batches). The code is as follows:

```
# initialize
>>> data = np.array([[1, 2, 3]],
...                    dtype=theano.config.floatX)
>>> x = T.fmatrix('x')
>>> w = theano.shared(np.asarray([[0.0, 0.0, 0.0]],
...                               dtype=theano.config.floatX))
>>> z = x.dot(w.T)
>>> update = [[w, w + 1.0]]

# compile
>>> net_input = theano.function(inputs=[],
...                             updates=update,
...                             givens={x: data},
...                             outputs=z)

# execute
>>> for i in range(5):
...     print('z:', net_input())
z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

Looking at the preceding code example, we also see that the `givens` attribute is a Python dictionary that maps a variable name to the actual Python object. Here, we set this name when we defined the `fmatrix`.

Wrapping things up – a linear regression example

Now that we familiarized ourselves with Theano, let's take a look at a really practical example and implement **Ordinary Least Squares** (**OLS**) regression. For a quick refresher on regression analysis, please refer to *Chapter 10, Predicting Continuous Target Variables with Regression Analysis*.

Let's start by creating a small one-dimensional toy dataset with ten training samples:

```
>>> X_train = np.asarray([[0.0], [1.0],
...                       [2.0], [3.0],
...                       [4.0], [5.0],
...                       [6.0], [7.0],
...                       [8.0], [9.0]],
...                      dtype=theano.config.floatX)
>>> y_train = np.asarray([1.0, 1.3,
...                       3.1, 2.0,
...                       5.0, 6.3,
...                       6.6, 7.4,
...                       8.0, 9.0],
...                      dtype=theano.config.floatX)
```

Note that we are using `theano.config.floatX` when we construct the NumPy arrays, so we can optionally toggle back and forth between CPU and GPU if we want.

Next, let's implement a training function to learn the weights of the linear regression model, using the sum of squared errors cost function. Note that w_0 is the bias unit (the y axis intercept at $x = 0$). The code is as follows:

```
import theano
from theano import tensor as T
import numpy as np

def train_linreg(X_train, y_train, eta, epochs):

    costs = []
    # Initialize arrays
    eta0 = T.fscalar('eta0')
    y = T.fvector(name='y')
    X = T.fmatrix(name='X')
```

```
w = theano.shared(np.zeros(
                      shape=(X_train.shape[1] + 1),
                      dtype=theano.config.floatX),
                   name='w')

# calculate cost
net_input = T.dot(X, w[1:]) + w[0]
errors = y - net_input
cost = T.sum(T.pow(errors, 2))

# perform gradient update
gradient = T.grad(cost, wrt=w)
update = [(w, w - eta0 * gradient)]

# compile model
train = theano.function(inputs=[eta0],
                        outputs=cost,
                        updates=update,
                        givens={X: X_train,
                                y: y_train,})

for _ in range(epochs):
    costs.append(train(eta))

return costs, w
```
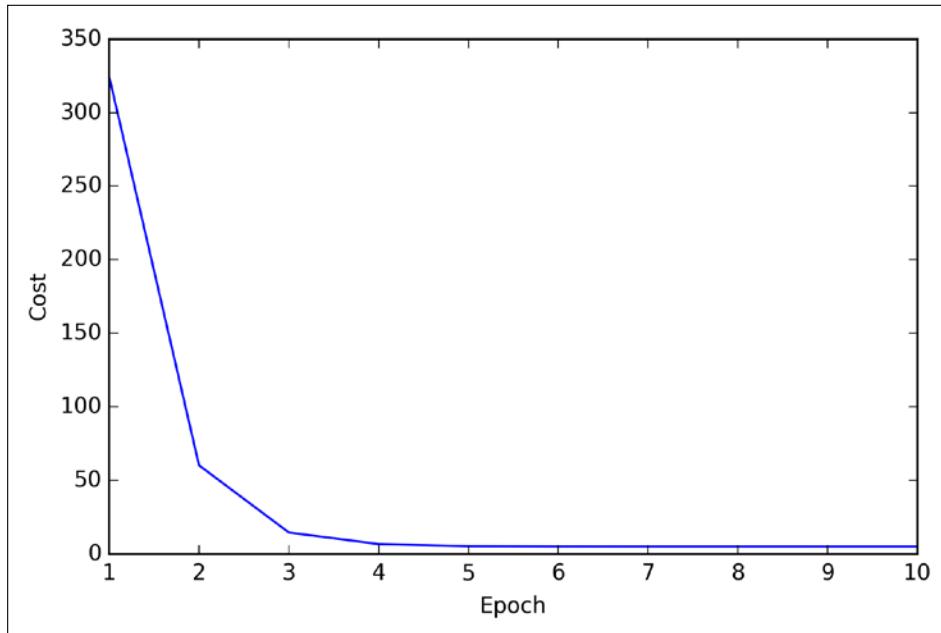
A really nice feature in Theano is the `grad` function that we used in the preceding code example. The `grad` function automatically computes the derivative of an expression *with respect to* its parameters that we passed to the function as the `wrt` argument.

After we implemented the training function, let's train our linear regression model and take a look at the values of the **Sum of Squared Errors** (**SSE**) cost function to check if it converged:

```
>>> import matplotlib.pyplot as plt
>>> costs, w = train_linreg(X_train, y_train, eta=0.001, epochs=10)
>>> plt.plot(range(1, len(costs)+1), costs)
>>> plt.tight_layout()
>>> plt.xlabel('Epoch')
>>> plt.ylabel('Cost')
>>> plt.show()
```

As we can see in the following plot, the learning algorithm already converged after the fifth epoch:

So far so good; by looking at the cost function, it seems that we built a working regression model from this particular dataset. Now, let's compile a new function to make predictions based on the input features:
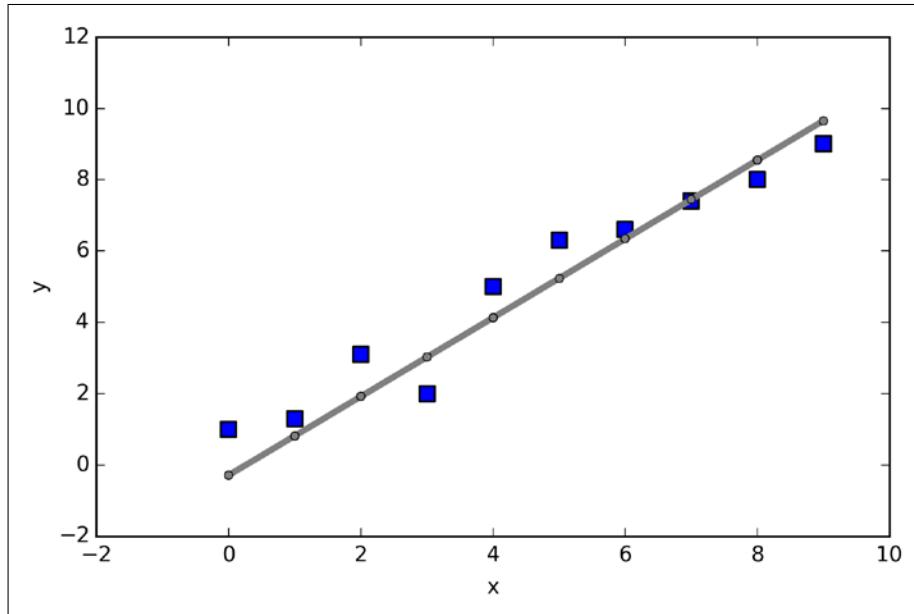
```
def predict_linreg(X, w):
    Xt = T.matrix(name='X')
    net_input = T.dot(Xt, w[1:]) + w[0]
    predict = theano.function(inputs=[Xt],
                              givens={w: w},
                              outputs=net_input)
    return predict(X)
```

Implementing a `predict` function was pretty straightforward following the three-step procedure of Theano: define, compile, and execute. Next, let's plot the linear regression fit on the training data:

```
>>> plt.scatter(X_train,
...             y_train,
...             marker='s',
...             s=50)
>>> plt.plot(range(X_train.shape[0]),
```

```
...             predict_linreg(X_train, w),
...             color='gray',
...             marker='o',
...             markersize=4,
...             linewidth=3)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

As we can see in the resulting plot, our model fits the data points appropriately:

Implementing a simple regression model was a good exercise to become familiar with the Theano API. However, our ultimate goal is to play out the advantages of Theano, that is, implementing powerful artificial neural networks. We should now be equipped with all the tools we would need to implement the multilayer perceptron from *Chapter 12*, *Training Artificial Neural Networks for Image Recognition*, in Theano. However, this would be rather boring, right? Thus, we will take a look at one of my favorite deep learning libraries built on top of Theano to make the experimentation with neural networks as convenient as possible. However, before we introduce the Keras library, let's first discuss the different choices of activation functions in neural networks in the next section.

Choosing activation functions for feedforward neural networks

For simplicity, we have only discussed the sigmoid activation function in context of multilayer feedforward neural networks so far; we used it in the hidden layer as well as the output layer in the multilayer perceptron implementation in *Chapter 12, Training Artificial Neural Networks for Image Recognition*. Although we referred to this activation function as *sigmoid* function—as it is commonly called in literature—the more precise definition would be *logistic function* or *negative log-likelihood function*. In the following subsections, you will learn more about alternative sigmoidal functions that are useful for implementing multilayer neural networks.

Technically, we could use any function as activation function in multilayer neural networks as long as it is differentiable. We could even use linear activation functions such as in Adaline (*Chapter 2, Training Machine Learning Algorithms for Classification*). However, in practice, it would not be very useful to use linear activation functions for both hidden and output layers, since we want to introduce nonlinearity in a typical artificial neural network to be able to tackle complex problem tasks. The sum of linear functions yields a linear function after all.

The logistic activation function that we used in the previous chapter probably mimics the concept of a neuron in a brain most closely: we can think of it as probability of whether a neuron fires or not. However, logistic activation functions can be problematic if we have highly negative inputs, since the output of the sigmoid function would be close to zero in this case. If the sigmoid function returns outputs that are close to zero, the neural network would learn very slowly and it becomes more likely that it gets trapped in local minima during training. This is why people often prefer a **hyperbolic tangent** as activation function in hidden layers. Before we discuss what a hyperbolic tangent looks like, let's briefly recapitulate some of the basics of the logistic function and look at a generalization that makes it more useful for multi-class classification tasks.

Logistic function recap

As we mentioned it in the introduction to this section, the logistic function, often just called the *sigmoid function*, is in fact a special case of a sigmoid function. We recall from the section on logistic regression in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, that we can use the logistic function to model the probability that sample x belongs to the positive class (class 1) in a binary classification task:

$$\phi_{logistic}(z) = \frac{1}{1+e^{-z}}$$

Here, the scalar variable z is defined as the net input:

$$z = w_0 x_0 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}$$

Note that w_0 is the bias unit (y-axis intercept, $x_0 = 1$). To provide a more concrete example, let's assume a model for a two-dimensional data point x and a model with the following weight coefficients assigned to the vector w:

```
>>> X = np.array([[1, 1.4, 1.5]])
>>> w = np.array([0.0, 0.2, 0.4])

>>> def net_input(X, w):
...     z = X.dot(w)
...     return z

>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))

>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)

>>> print('P(y=1|x) = %.3f'
...        % logistic_activation(X, w)[0])
P(y=1|x) = 0.707
```

If we calculate the net input and use it to activate a logistic neuron with those particular feature values and weight coefficients, we get back a value of 0.707, which we can interpret as a 70.7 percent probability that this particular sample x belongs to the positive class. In *Chapter 12*, *Training Artificial Neural Networks for Image Recognition*, we used the one-hot encoding technique to compute the values in the output layer consisting of multiple logistic activation units. However, as we will demonstrate with the following code example, an output layer consisting of multiple logistic activation units does not produce meaningful, interpretable probability values:

```
# W : array, shape = [n_output_units, n_hidden_units+1]
#         Weight matrix for hidden layer -> output layer.
# note that first column (A[:][0] = 1) are the bias units
>>> W = np.array([[1.1, 1.2, 1.3, 0.5],
...               [0.1, 0.2, 0.4, 0.1],
...               [0.2, 0.5, 2.1, 1.9]])

# A : array, shape = [n_hidden+1, n_samples]
#         Activation of hidden layer.
# note that first element (A[0][0] = 1) is the bias unit
>>> A = np.array([[1.0],
...               [0.1],
...               [0.3],
...               [0.7]])

# Z : array, shape = [n_output_units, n_samples]
#         Net input of the output layer.
>>> Z = W.dot(A)
>>> y_probas = logistic(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
 [[ 0.87653295]
 [ 0.57688526]
 [ 0.90114393]]
```

As we can see in the output, the probability that the particular sample belongs to the first class is almost 88 percent, the probability that the particular sample belongs to the second class is almost 58 percent, and the probability that the particular sample belongs to the third class is 90 percent, respectively. This is clearly confusing, since we all know that a percentage should intuitively be expressed as a fraction of 100. However, this is in fact not a big concern if we only use our model to predict the class labels, not the class membership probabilities.

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('predicted class label: %d' % y_class[0])
predicted class label: 2
```

However, in certain contexts, it can be useful to return meaningful class probabilities for multi-class predictions. In the next section, we will take a look at a generalization of the logistic function, the **softmax** function, which can help us with this task.

Estimating probabilities in multi-class classification via the softmax function

The **softmax** function is a generalization of the logistic function that allows us to compute meaningful class-probabilities in multi-class settings (multinomial logistic regression). In softmax, the probability of a particular sample with net input z belongs to the i th class can be computed with a normalization term in the denominator that is the sum of all M linear functions:

$$P\left(y = i \mid z\right) = \phi_{softmax}\left(z\right) = \frac{e_i^z}{\sum_{m=1}^{M} e_m^z}$$

To see softmax in action, let's code it up in Python:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))

>>> def softmax_activation(X, w):
...     z = net_input(X, w)
...     return softmax(z)

>>> y_probas = softmax(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
 [[ 0.40386493]
 [ 0.07756222]
 [ 0.51857284]]
>>> y_probas.sum()
1.0
```

As we can see, the predicted class probabilities now sum up to one, as we would expect. It is also notable that the probability for the second class is close to zero, since there is a large gap between z_1 and $\max(z)$. However, note that the predicted class label is the same as in the logistic function. Intuitively, it may help to think of the softmax function as a *normalized* logistic function that is useful to obtain meaningful class-membership predictions in multi-class settings.

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('predicted class label:
...          %d' % y_class[0])
predicted class label: 2
```

Broadening the output spectrum by using a hyperbolic tangent

Another sigmoid function that is often used in the hidden layers of artificial neural networks is the **hyperbolic tangent** (**tanh**), which can be interpreted as a rescaled version of the logistic function.

$$\phi_{\text{tanh}}(z) = 2 \times \phi_{logistic}(2 \times z) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\phi_{logistic}(z) = \frac{1}{1 + e^{-z}}$$

The advantage of the hyperbolic tangent over the logistic function is that it has a broader output spectrum and ranges the open interval (-1, 1), which can improve the convergence of the back propagation algorithm (C. M. Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995, pp. 500-501). In contrast, the logistic function returns an output signal that ranges the open interval (0, 1). For an intuitive comparison of the logistic function and the hyperbolic tangent, let's plot the two sigmoid functions:

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

>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)

>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)

>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('net input $z$')
>>> plt.ylabel('activation $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle='--')
>>> plt.axhline(0.5, color='black', linestyle='--')
>>> plt.axhline(0, color='black', linestyle='--')
>>> plt.axhline(-1, color='black', linestyle='--')

>>> plt.plot(z, tanh_act,
...          linewidth=2,
...          color='black',
...          label='tanh')
>>> plt.plot(z, log_act,
...          linewidth=2,
...          color='lightgreen',
...          label='logistic')

>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```
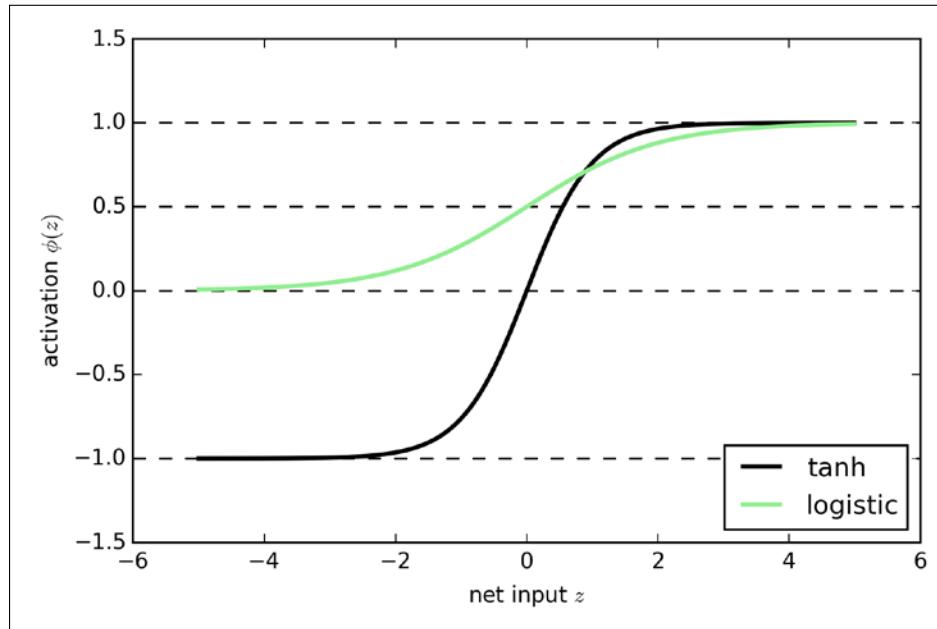
As we can see, the shapes of the two sigmoidal curves look very similar; however, the `tanh` function has 2x larger output space than the logistic function:
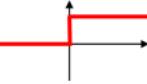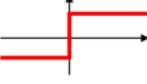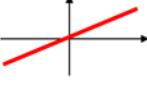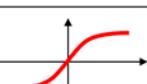
Note that we implemented the `logistic` and `tanh` functions verbosely for the purpose of illustration. In practice, we can use NumPy's `tanh` function to achieve the same results:

```
>>>   tanh_act = np.tanh(z)
```

In addition, the `logistic` function is available in SciPy's `special` module:

```
>>> from scipy.special import expit
>>> log_act = expit(z)
```

Now that we know more about the different activation functions that are commonly used in artificial neural networks, let's conclude this section with an overview of the different activation functions that we encountered in this book.

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \dfrac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

Training neural networks efficiently using Keras

In this section, we will take a look at Keras, one of the most recently developed libraries to facilitate neural network training. The development on Keras started in the early months of 2015; as of today, it has evolved into one of the most popular and widely used libraries that are built on top of Theano, and allows us to utilize our GPU to accelerate neural network training. One of its prominent features is that it's a very intuitive API, which allows us to implement neural networks in only a few lines of code. Once you have Theano installed, you can install Keras from PyPI by executing the following command from your terminal command line:

```
pip install Keras
```

For more information about Keras, please visit the official website at `http://keras.io`.

To see what neural network training via Keras looks like, let's implement a multilayer perceptron to classify the handwritten digits from the MNIST dataset, which we introduced in the previous chapter. The MNIST dataset can be downloaded from `http://yann.lecun.com/exdb/mnist/` in four parts as listed here:

- `train-images-idx3-ubyte.gz`: These are training set images (9912422 bytes)
- `train-labels-idx1-ubyte.gz`: These are training set labels (28881 bytes)
- `t10k-images-idx3-ubyte.gz`: These are test set images (1648877 bytes)
- `t10k-labels-idx1-ubyte.gz`: These are test set labels (4542 bytes)

After downloading and unzipping the archives, we place the files into a directory `mnist` in our current working directory, so that we can load the training as well as the test dataset using the following function:

```python
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                                % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                                % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                             dtype=np.uint8).reshape(len(labels), 784)
```

```
      return images, labels
X_train, y_train = load_mnist('mnist', kind='train')
print('Rows: %d, columns: %d' % (X_train.shape[0], X_train.shape[1]))
Rows: 60000, columns: 784
X_test, y_test = load_mnist('mnist', kind='t10k')
print('Rows: %d, columns: %d' % (X_test.shape[0], X_test.shape[1]))
Rows: 10000, columns: 784
```

On the following pages, we will walk through the code examples for using Keras step by step, which you can directly execute from your Python interpreter. However, if you are interested in training the neural network on your GPU, you can either put it into a Python script, or download the respective code from the Packt Publishing website. In order to run the Python script on your GPU, execute the following command from the directory where the `mnist_keras_mlp.py` file is located:

```
 THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python mnist_
keras_mlp.py
```

To continue with the preparation of the training data, let's cast the MNIST image array into 32-bit format:

```
>>> import theano
>>> theano.config.floatX = 'float32'
>>> X_train = X_train.astype(theano.config.floatX)
>>> X_test = X_test.astype(theano.config.floatX)
```

Next, we need to convert the class labels (integers 0-9) into the one-hot format. Fortunately, Keras provides a convenient tool for this:

```
>>> from keras.utils import np_utils
>>> print('First 3 labels: ', y_train[:3])
First 3 labels:  [5 0 4]
>>> y_train_ohe = np_utils.to_categorical(y_train)
>>> print('\nFirst 3 labels (one-hot):\n', y_train_ohe[:3])
First 3 labels (one-hot):
 [[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]]
```

Now, we can get to the interesting part and implement a neural network. Here, we will use the same architecture as in *Chapter 12, Training Artificial Neural Networks for Image Recognition*. However, we will replace the logistic units in the hidden layer with hyperbolic tangent activation functions, replace the logistic function in the output layer with softmax, and add an additional hidden layer. Keras makes these tasks very simple, as you can see in the following code implementation:

```
>>> from keras.models import Sequential
>>> from keras.layers.core import Dense
>>> from keras.optimizers import SGD

>>> np.random.seed(1)

>>> model = Sequential()
>>> model.add(Dense(input_dim=X_train.shape[1],
...                  output_dim=50,
...                  init='uniform',
...                  activation='tanh'))

>>> model.add(Dense(input_dim=50,
...                  output_dim=50,
...                  init='uniform',
...                  activation='tanh'))

>>> model.add(Dense(input_dim=50,
...                  output_dim=y_train_ohe.shape[1],
...                  init='uniform',
...                  activation='softmax'))

>>> sgd = SGD(lr=0.001, decay=1e-7, momentum=.9)
>>> model.compile(loss='categorical_crossentropy', optimizer=sgd)
```

First, we initialize a new model using the `Sequential` class to implement a feedforward neural network. Then, we can add as many layers to it as we like. However, since the first layer that we add is the input layer, we have to make sure that the `input_dim` attribute matches the number of features (columns) in the training set (here, 768). Also, we have to make sure that the number of output units (`output_dim`) and input units (`input_dim`) of two consecutive layers match. In the preceding example, we added two hidden layers with 50 hidden units plus 1 bias unit each. Note that bias units are initialized to 0 in fully connected networks in Keras. This is in contrast to the MLP implementation in *Chapter 12, Training Artificial Neural Networks for Image Recognition*, where we initialized the bias units to 1, which is a more common (not necessarily better) convention.

Finally, the number of units in the output layer should be equal to the number of unique class labels—the number of columns in the one-hot encoded class label array. Before we can compile our model, we also have to define an optimizer. In the preceding example, we chose a stochastic gradient descent optimization, which we are already familiar with, from previous chapters. Furthermore, we can set values for the weight decay constant and momentum learning to adjust the learning rate at each epoch as discussed in *Chapter 12, Training Artificial Neural Networks for Image Recognition*. Lastly, we set the cost (or loss) function to `categorical_crossentropy`. The (binary) cross-entropy is just the technical term for the cost function in logistic regression, and the categorical cross-entropy is its generalization for multi-class predictions via softmax. After compiling the model, we can now train it by calling the `fit` method. Here, we are using mini-batch stochastic gradient with a batch size of 300 training samples per batch. We train the MLP over 50 epochs, and we can follow the optimization of the `cost` function during training by setting `verbose=1`. The `validation_split` parameter is especially handy, since it will reserve 10 percent of the training data (here, 6,000 samples) for validation after each epoch, so that we can check if the model is overfitting during training.

```
>>> model.fit(X_train,
...           y_train_ohe,
...           nb_epoch=50,
...           batch_size=300,
...           verbose=1,
...           validation_split=0.1,
...           show_accuracy=True)
Train on 54000 samples, validate on 6000 samples
Epoch 0
54000/54000 [==============================] - 1s - loss: 2.2290 -
acc: 0.3592 - val_loss: 2.1094 - val_acc: 0.5342
Epoch 1
54000/54000 [==============================] - 1s - loss: 1.8850 -
acc: 0.5279 - val_loss: 1.6098 - val_acc: 0.5617
Epoch 2
54000/54000 [==============================] - 1s - loss: 1.3903 -
acc: 0.5884 - val_loss: 1.1666 - val_acc: 0.6707
Epoch 3
54000/54000 [==============================] - 1s - loss: 1.0592 -
acc: 0.6936 - val_loss: 0.8961 - val_acc: 0.7615
 [...]
Epoch 49
54000/54000 [==============================] - 1s - loss: 0.1907 -
acc: 0.9432 - val_loss: 0.1749 - val_acc: 0.9482
```

Printing the value of the cost function is extremely useful during training, since we can quickly spot whether the cost is decreasing during training and stop the algorithm earlier if otherwise to tune the hyperparameter values.

To predict the class labels, we can then use the `predict_classes` method to return the class labels directly as integers:

```
>>> y_train_pred = model.predict_classes(X_train, verbose=0)
>>> print('First 3 predictions: ', y_train_pred[:3])
>>> First 3 predictions:  [5 0 4]
```

Finally, let's print the model accuracy on training and test sets:

```
>>> train_acc = np.sum(
...        y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (train_acc * 100))
Training accuracy: 94.51%

>>> y_test_pred = model.predict_classes(X_test, verbose=0)
>>> test_acc = np.sum(y_test == y_test_pred,
...                     axis=0) / X_test.shape[0]
print('Test accuracy: %.2f%%' % (test_acc * 100))
Test accuracy: 94.39%
```

Note that this is just a very simple neural network without optimized tuning parameters. If you are interested in playing more with Keras, please feel free to further tweak the learning rate, momentum, weight decay, and number of hidden units.

> Although Keras is a great library for implementing and experimenting with neural networks, there are many other Theano wrapper libraries that are worth mentioning. A prominent example is Pylearn2 (`http://deeplearning.net/software/pylearn2/`), which has been developed in the LISA lab in Montreal. Also, Lasagne (`https://github.com/Lasagne/Lasagne`) may be of interest to you if you prefer a more minimalistic but extensible library, that offers more control over the underlying Theano code.

Summary

I hope you enjoyed this last chapter of an exciting tour of machine learning. Throughout this book, we covered all of the essential topics that this field has to offer, and you should now be well equipped to put those techniques into action to solve real-world problems.

We started our journey with a brief overview of the different types of learning tasks: supervised learning, reinforcement learning, and unsupervised learning. We discussed several different learning algorithms that can be used for classification, starting with simple single-layer neural networks in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. Then, we discussed more advanced classification algorithms in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, and you learned about the most important aspects of a machine learning pipeline in *Chapter 4*, *Building Good Training Sets – Data Preprocessing* and *Chapter 5*, *Compressing Data via Dimensionality Reduction*. Remember that even the most advanced algorithm is limited by the information in the training data that it gets to learn from. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, you learned about the best practices to build and evaluate predictive models, which is another important aspect in machine learning applications. If one single learning algorithm does not achieve the performance we desire, it can sometimes be helpful to create an ensemble of experts to make a prediction. We discussed this in *Chapter 7*, *Combining Different Models for Ensemble Learning*. In *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*, we applied machine learning to analyze the probably most interesting form of data in the modern age that is dominated by social media platforms on the Internet: text documents. However, machine learning techniques are not limited to offline data analysis, and in *Chapter 9*, *Embedding a Machine Learning Model into a Web Application*, we saw how to embed a machine learning model into a web application to share it with the outside world. For the most part, our focus was on algorithms for classification, probably the most popular application of machine learning. However, this is not where it ends! In *Chapter 10*, *Predicting Continuous Target Variables with Regression Analysis*, we explored several algorithms for regression analysis to predict continuous-valued output values. Another exciting subfield of machine learning is clustering analysis, which can help us to find hidden structures in data even if our training data does not come with the right answers to learn from. We discussed this in *Chapter 11*, *Working with Unlabeled Data – Clustering Analysis*.

In the last two chapters of this book, we caught a glimpse of the most beautiful and most exciting algorithms in the whole machine learning field: artificial neural networks. Although deep learning really is beyond the scope of this book, I hope I could at least kindle your interest to follow the most recent advancement in this field. If you are considering a career as a machine learning researcher, or even if you just want to keep up to date with the current advancement in this field, I can recommend you to follow the works of the leading experts in this field, such as Geoff Hinton (`http://www.cs.toronto.edu/~hinton/`), Andrew Ng (`http://www.andrewng.org`), Yann LeCun (`http://yann.lecun.com`), Juergen Schmidhuber (`http://people.idsia.ch/~juergen/`), and Yoshua Bengio (`http://www.iro.umontreal.ca/~bengioy`), just to name a few. Also, please do not hesitate to join the scikit-learn, Theano, and Keras mailing lists to participate in interesting discussions around these libraries, and machine learning in general. I am looking forward to meeting you there! You are always welcome to contact me if you have any questions about this book or need some general tips about machine learning.

I hope this journey through the different aspects of machine learning was really worthwhile, and you learned many new and useful skills to advance your career and apply them to real-world problem solving.

Module 2

Designing Machine Learning Systems with Python

Leverage benefits of machine learning techniques using Python

1
Thinking in Machine Learning

Machine learning systems have a profound and exciting ability to provide important insights to an amazing variety of applications; from groundbreaking and life-saving medical research, to discovering fundamental physical aspects of our universe. From providing us with better, cleaner food, to web analytics and economic modeling. In fact, there are hardly any areas of our lives that have not been touched by this technology in some way. With an expanding Internet of Things, there is a staggering amount of data being generated, and it is clear that intelligent systems are changing societies in quite dramatic ways. With open source tools, such as those provided by Python and its libraries, and the increasing open source knowledge base represented by the Web, it is relatively easy and cheap to learn and apply this technology in new and exciting ways. In this chapter, we will cover the following topics:

- Human interface
- Design principles
- Models
- Unified modelling language

The human interface

For those of you old enough, or unfortunate enough, to have used early versions of the Microsoft office suite, you will probably remember the Mr Clippy office assistant. This feature, first introduced in Office 97, popped up uninvited from the bottom right-hand side of your computer screen every time you typed the word 'Dear' at the beginning of a document, with the prompt "it looks like you are writing a letter, would you like help with that?".

Mr Clippy, turned on by default in early versions of Office, was almost universally derided by users of the software and could go down in history as one of machine learning's first big fails.

So, why was the cheery Mr Clippy so hated? Clearly the folks at Microsoft, at the forefront of consumer software development, were not stupid, and the idea that an automated assistant could help with day to day office tasks is not necessarily a bad idea. Indeed, later incarnations of automated assistants, the best ones at least, operate seamlessly in the background and provide a demonstrable increase in work efficiency. Consider predictive text. There are many examples, some very funny, of where predictive text has gone spectacularly wrong, but in the majority of cases where it doesn't fail, it goes unnoticed. It just becomes part of our normal work flow.

At this point, we need a distinction between error and failure. Mr Clippy failed because it was obtrusive and poorly designed, not necessarily because it was in error; that is, it could make the right suggestion, but chances are you already know that you are writing a letter. Predictive text has a high error rate, that is, it often gets the prediction wrong, but it does not fail largely because of the way it is designed to fail: unobtrusively.

The design of any system that has a *tightly coupled human interface*, to use systems engineering speak, is difficult. Human behavior, like the natural world in general, is not something we can always predict. Expression recognition systems, natural language processing, and gesture recognition technology, amongst other things, all open up new ways of human-machine interaction, and this has important applications for the machine learning specialist.

Whenever we are designing a system that requires human input, we need to anticipate the possible ways, not just the intended ways, a human will interact with the system. In essence, what we are trying to do with these systems is to instil in them some understanding of the broad panorama of human experience.

In the first years of the web, search engines used a simple system based on the number of times search terms appeared in articles. Web developers soon began gaming the system by increasing the number of key search terms. Clearly, this would lead to a keyword arms race and result in a very boring web. The page rank system measuring the number of quality inbound links was designed to provide a more accurate search result. Now, of course, modern search engines use more sophisticated and secret algorithms.
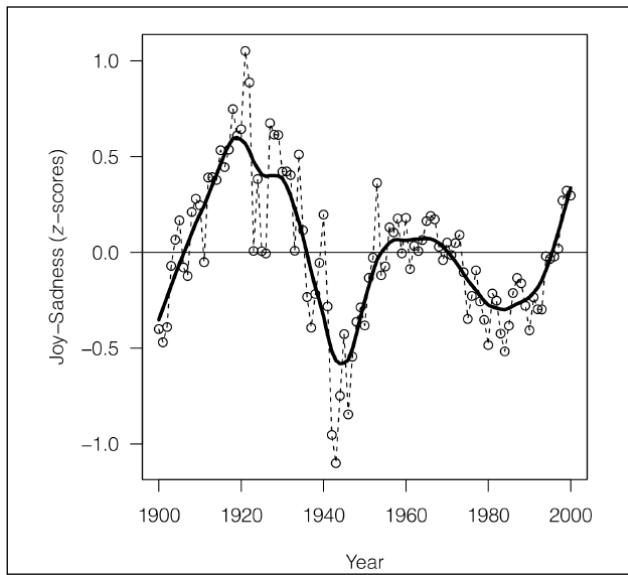
What is also important for ML designers is the ever increasing amount of data that is being generated. This presents several challenges, most notably its sheer vastness. However, the power of algorithms in extracting knowledge and insights that would not have been possible with smaller data sets is massive. So, many human interactions are now digitized, and we are only just beginning to understand and explore the many ways in which this data can be used.

As a curious example, consider the study *The expression of emotion in 20th century books* (Acerbi et al, 2013). Though strictly more of a data analysis study, rather than machine learning, it is illustrative for several reasons. Its purpose was to chart the emotional content, in terms of a mood score, of text extracted from books of the 20th century. With access to a large volume of digitized text through the project Gutenberg digital library, WordNet (`http://wordnet.princeton.edu/wordnet/`), and Google's **Ngram** database (`books.google.com/ngrams`), the authors of this study were able to map cultural change over the 20th century as reflected in the literature of the time. They did this by mapping trends in the usage of the *mood* words.

For this study, the authors labeled each word (*a 1gram*) and associated it with a mood score and the year it was published. We can see that emotion words, such as joy, sadness, fear, and so forth, can be scored according to the positive or negative mood they evoke. The mood score was obtained from WordNet (`wordnet.princeton.edu`). WordNet assigns an affect score to each mood word. Finally, the authors simply counted the occurrences of each mood word:

$$M = \frac{1}{n}\sum_{i=1}^{n}\frac{c_i}{C_{the}} \quad M_z = \frac{M - \mu_M}{\sigma_M}$$

Here, *ci* is the count of a particular mood word, *n* is the total count of mood words (not all words, just words with a mood score), and C_{the} is the count of the word *the* in the text. This normalizes the sum to take into account that some years more books were written (or digitized). Also, since many later books tend to contain more technical language, the word *the* was used to normalize rather than get the total word count. This gives a more accurate representation of emotion over a long time period in prose text. Finally, the score is normalized according to a normal distribution, M_z, by subtracting the mean and dividing by the standard deviation.

This figure is taken from *The expression of Emotions in 20th Century Books, (Alberto Acerbi, Vasileios Lampos, Phillip Garnett, R. Alexander Bentley) PLOS.*

Here we can see one of the graphs generated by this study. It shows the joy-sadness score for books written in this period, and clearly shows a negative trend associated with the period of World War II.

This study is interesting for several reasons. Firstly, it is an example of data-driven science, where previously considered *soft* sciences, such as sociology and anthropology, are given a solid empirical footing. Despite some pretty impressive results, this study was relatively easy to implement. This is mainly because most of the hard work had already been done by WordNet and Google. This highlights how using data resources that are freely available on the Internet, and software tools such as the Python's data and machine learning packages, anyone with the data skills and motivation can build on this work.

Design principles

An analogy is often made between systems design and designing other things such as a house. To a certain extent, this analogy holds true. We are attempting to place design components into a structure that meets a specification. The analogy breaks down when we consider their respective operating environments. It is generally assumed in the design of a house that the landscape, once suitably formed, will not change.

Software environments are slightly different. Systems are interactive and dynamic. Any system that we design will be nested inside other systems, either electronic, physical, or human. In the same way different layers in computer networks (application layer, transport layer, physical layer, and so on) nest different sets of meanings and function, so to do activities performed at different levels of a project.

As the designer of these systems, we must also have a strong awareness of the setting, that is, the domain in which we work. This knowledge gives us clues to patterns in our data and helps us give context to our work.

Machine learning projects can be divided into five distinct activities, shown as follows:

- Defining the object and specification
- Preparing and exploring the data
- Model building
- Implementation
- Testing
- Deployment

The designer is mainly concerned with the first three. However, they often play, and in many projects must play, a major role in other activities. It should also be said that a project's timeline is not necessarily a linear sequence of these activities. The important point is that they are distinct activities. They may occur in parallel to each other, and in other ways interact with each other, but they generally involve different types of tasks that can be separated in terms of human and other resources, the stage of the project, and externalities. Also, we need to consider that different activities involve distinct operational modes. Consider the different ways in which your brain works when you are sketching out an idea, as compared to when you are working on a specific analytical task, say a piece of code.

Often, the hardest question is where to begin. We can start drilling into the different elements of a problem, with an idea of a feature set and perhaps an idea of the model or models we might use. This may lead to a defined object and specification, or we may have to do some preliminary research such as checking possible data sets and sources, available technologies, or talking to other engineers, technicians, and users of the system. We need to explore the operating environment and the various constraints; is it part of a web application, or is it a laboratory research tool for scientists?

In the early stages of design, our work flow will flip between working on the different elements. For instance, we start with a general problem—perhaps having an idea of the task, or tasks, necessary to solve it—then we divide it into what we think are the key features, try it out on a few models with a toy dataset, go back to refine the feature set, adjust our model, precisely define tasks, and refine the model. When we feel our system is robust enough, we can test it out on some real data. Of course, then we may need to go back and change our feature set.

Selecting and optimizing features is often a major activity (really, a task in itself) for the machine learning designer. We cannot really decide what features we need until we have adequately described the task, and of course, both the task and features are constrained by the types of feasible models we can build.

Types of questions

As designers, we are asked to solve a problem. We are given some data and an expected output. The first step is to frame the problem in a way that a machine can understand it, and in a way that carries meaning for a human. The following six broad approaches are what we can take to precisely define our machine learning problem:

- **Exploratory**: Here, we analyze data, looking for patterns such as a trend or relationship between variables. Exploration will often lead to a hypothesis such as linking diet with disease, or crime rate with urban dwellings.

- **Descriptive**: Here, we try to summarize specific features of our data. For instance, the average life expectancy, average temperature, or the number of left-handed people in a population.

- **Inferential**: An inferential question is one that attempts to support a hypothesis, for instance, proving (or disproving) a general link between life expectancy and income by using different data sets.

- **Predictive**: Here, we are trying to anticipate future behavior. For instance, predicting life expectancy by analyzing income.

- **Casual**: This is an attempt to find out what causes something. Does low income cause a lower life expectancy?

- **Mechanistic**: This tries to answer questions such as "what are the mechanisms that link income with life expectancy?"

Most machine learning problems involve several of these types of questions during development. For instance, we may first explore the data looking for patterns or trends, and then we may describe certain key features of our data. This may enable us to make a prediction, and find a cause or a mechanism behind a particular problem.

Are you asking the right question?

The question must be plausible and meaningful in its subject area. This domain knowledge enables you to understand the things that are important in your data and to see where a certain pattern or correlation has meaning.

The question should be as specific as possible, while still giving a meaningful answer. It is common for it to begin as a generalized statement, such as "I wonder if wealthy means healthy". So, you do some further research and find you can get statistics for wealth by geographic region, say from the tax office. We can measure health through its inverse, that is, illness, say by hospital admissions, and we can test our initial proposition, "wealthy means healthy", by tying illness to geographic region. We can see that a more specific question relies on several, perhaps questionable, assumptions.

We should also consider that our results may be confounded by the fact that poorer people may not have healthcare insurance, so are less likely to go to a hospital despite illness. There is an interaction between what we want to find out and what we are trying to measure. This interaction perhaps hides a true rate of illness. All is not lost, however. Because we know about these things, then perhaps we can account for them in our model.

We can make things a lot easier by learning as much as we can about the domain we are working in.

You could possibly save yourself a lot of time by checking whether the question you are asking, or part of it, has already been answered, or if there are data sets available that may shed some light on that topic. Often, you have to approach a problem from several different angles at once. Do as much preparatory research as you can. It is quite likely that other designers have done work that could shed light on your own.

Tasks
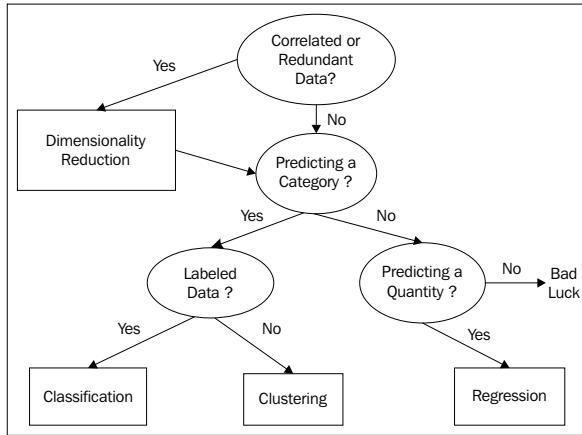
A task is a specific activity conducted over a period of time. We have to distinguish between the human tasks (planning, designing, and implementing) to the machine tasks (classification, clustering, regression, and so on). Also consider when there is overlap between human and machine, for example, as in selecting features for a model. Our true goal in machine learning is to transform as many of these tasks as we can from human tasks to machine tasks.

It is not always easy to match a real world problem to a specific task. Many real world problems may seem to be conceptually linked but require a very different solution. Alternatively, problems that appear completely different may require similar methods. Unfortunately, there is no simple lookup table to match a particular task to a problem. A lot depends on the setting and domain. A similar problem in one domain may be unsolvable in another, perhaps because of lack of data. There are, however, a small number of tasks that are applied to a large number of methods to solve many of the most common problem types. In other words, in the space of all possible programming tasks, there is a subset of tasks that are useful to our particular problem. Within this subset, there is a smaller subset of tasks that are easy and can actually be applied usefully to our problem.

Machine learning tasks occur in three broad settings:

- **Supervised learning**: The goal here is to learn a model from labeled training data that allows predictions to be made on unseen future data.

- **Unsupervised learning**: Here we deal with unlabeled data and our goal is to find hidden patterns in this data to extract meaningful information.

- **Reinforcement learning**: The goal here is to develop a system that improves its performance based on the interactions it has with its environment. This usually involves a reward signal. This is similar to supervised learning, except that rather than having a labeled training set, reinforcement learning uses a reward function to continually improve its performance.

Now, let's take a look at some of the major machine learning tasks. The following diagram should give you a starting point to try and decide what type of task is appropriate for different machine learning problems:

Classification

Classification is probably the most common type of task; this is due in part to the fact that it is relatively easy, well understood, and solves a lot of common problems. Classification is about assigning classes to a set of instances, based on their features. This is a supervised learning method because it relies on a labeled training set to learn a set of model parameters. This model can then be applied to unlabeled data to make a prediction on what class each instance belongs to. There are broadly two types of classification tasks: **binary classification** and **multiclass classification**. A typical binary classification task is e-mail spam detection. Here we use the contents of an e-mail to determine if it belongs to one of the two classes: spam or not spam. An example of multiclass classification is handwriting recognition, where we try to predict a class, for example, the letter name. In this case, we have one class for each of the alpha numeric characters. Multiclass classification can sometimes be achieved by chaining binary classification tasks together, however, we lose information this way, and we are unable to define a single decision boundary. For this reason, multiclass classification is often treated separately from binary classification.

Regression

There are cases where what we are interested in are not discrete classes, but a continuous variable, for instance, a probability. These types of problems are regression problems. The aim of regression analysis is to understand how changes to the input, independent variables, effect changes to the dependent variable. The simplest regression problems are linear and involve fitting a straight line to a set of data in order to make a prediction. This is usually done by minimizing the sum of squared errors in each instance in the training set. Typical regression problems include estimating the likelihood of a disease given a range and severity of symptoms, or predicting test scores given past performance.

Clustering

Clustering is the most well known unsupervised method. Here, we are concerned with making a measurement of similarity between instances in an unlabeled dataset. We often use geometric models to determine the distance between instances, based on their feature values. We can use an arbitrary measurement of closeness to determine what cluster each instance belongs to. Clustering is often used in data mining and exploratory data analysis. There are a large variety of methods and algorithms that perform this task, and some of the approaches include the distance-based method, as well as finding a center point for each cluster, or using statistical techniques based on distributions.

Related to clustering is association; this is an unsupervised task to find a certain type of pattern in the data. This task is behind product recommender systems such as those provided by Amazon and other on-line shops.

Dimensionality reduction

Many data sets contain a large number of features or measurements associated with each instance. This can present a challenge in terms of computational power and memory allocation. Also many features may contain redundant information or information that is correlated to other features. In these cases, the performance of our learning model may be significantly degraded. Dimensionality reduction is most often used in feature prepossessing; it compresses the data into a lower dimension sub space while retaining useful information. Dimensionality reduction is also used when we want to visualize data, typically by projecting higher dimensions onto one, two, or three dimensions.

From these basic machine tasks, there are a number of derived tasks. In many applications, this may simply be applying the learning model to a prediction to establish a casual relationship. We must remember that explaining and predicting are not the same. A model can make a prediction, but unless we know explicitly how it made the prediction, we cannot begin to form a comprehensible explanation. An explanation requires human knowledge of the domain.

We can also use a prediction model to find exceptions from a general pattern. Here we are interested in the individual cases that deviate from the predictions. This is often called **anomaly detection** and has wide applications in things like detecting bank fraud, noise filtering, and even in the search for extraterrestrial life.

An important and potentially useful task is subgroup discovery. Our goal here is not, as in clustering, to partition the entire domain, but rather to find a subgroup that has a substantially different distribution. In essence, subgroup discovery is trying to find relationships between a dependent target variables and many independent explaining variables. We are not trying to find a complete relationship, but rather a group of instances that are different in ways that are important to the domain. For instance, establishing a subgroup, *smoker = true* and *family history = true* for a target variable of *heart disease = true*.

Finally, we consider control type tasks. These act to optimize control settings to maximize a payoff, given different conditions. This can be achieved in several ways. We can clone expert behavior: the machine learns directly from a human and makes predictions on actions given different conditions. The task is to learn a prediction model for the expert's actions. This is similar to reinforcement learning, where the task is to learn a relationship between conditions and optimal action.

Errors

In machine learning systems, software flaws can have very serious real world consequences; what happens if your algorithm, embedded in an assembly line robot, classifies a human as a production component? Clearly, in critical systems, you need to plan for failure. There should be a robust fault and error detection procedure embedded in your design process and systems.

Sometimes it is necessary to design very complex systems simply for the purpose of debugging and checking for logic flaws. It may be necessary to generate data sets with specific statistical structures, or create *artificial humans* to mimic an interface. For example, developing a methodology to verify that the logic of your design is sound at the data, model, and task levels. Errors can be hard to track, and as a scientist, you must assume that there are errors and try to prove otherwise.

The idea of recognizing and gracefully catching errors is important for the software designer, but as machine learning systems designers, we must take it a step further. We need to be able to capture, in our models, the ability to learn from an error.

Consideration must be given to how we select our test set, and in particular, how representative it is of the rest of the dataset. For instance, if it is noisy compared to the training set, it will give poor results on the test set, suggesting that our model is overfitting, when in fact, this is not the case. To avoid this, a process of cross validation is used. This works by randomly dividing the data into, for example, ten chunks of equal size. We use nine chunks for training the model and one for testing. We do this 10 times, using each chunk once for testing. Finally, we take an average of test set performance. Cross validation is used with other supervised learning problems besides classification, but as you would expect, unsupervised learning problems need to be evaluated differently.

With an unsupervised task we do not have a labeled training set. Evaluation can therefore be a little tricky since we do not know what a correct answer looks like. In a clustering problem, for instance, we can compare the quality of different models by measures such as the ratio of cluster diameter compared to the distance between clusters. However, in problems of any complexity, we can never tell if there is another model, not yet built, which is better.

Optimization

Optimization problems are ubiquitous in many different domains, such as finance, business, management, sciences, mathematics, and engineering. Optimization problems consist of the following:

- An objective function that we want to maximize or minimize.
- Decision variables, that is, a set of controllable inputs. These inputs are varied within the specified constraints in order to satisfy the objective function.
- Parameters, which are uncontrollable or fixed inputs.
- Constraints are relations between decision variables and parameters. They define what values the decision variables can have.

Most optimization problems have a single objective function. In the cases where we may have multiple objective functions, we often find that they conflict with each other, for example, reducing costs and increasing output. In practice, we try to reformulate multiple objectives into a single function, perhaps by creating a weighted combination of objective functions. In our costs and output example, a variable along the lines of cost per unit might work.

The decision variables are the variables we control to achieve the objective. They may include things such as resources or labor. The parameters of the module are fixed for each run of the model. We may use several *cases*, where we choose different parameters to test variations in multiple conditions.

There are literally thousands of solution algorithms to the many different types of optimization problems. Most of them involve first finding a feasible solution, then iteratively improving on it by adjusting the decision variables to hopefully find an optimum solution. Many optimization problems can be solved reasonably well with linear programming techniques. They assume that the objective function and all the constraints are linear with respect to the decision variables. Where these relationships are not linear, we often use a suitable quadratic function. If the system is non-linear, then the objective function may not be convex. That is, it may have more than one local minima, and there is no assurance that a local minima is a global minima.

Linear programming

Why are linear models so ubiquitous? Firstly, they are relatively easy to understand and implement. They are based on a well founded mathematical theory that was developed around the mid 1700s and that later played a pivotal role in the development of the digital computer. Computers are uniquely tasked to implement linear programs because computers were conceptualized largely on the basis of the theory of linear programming. Linear functions are always convex, meaning they have only one minima. **Linear Programming** (**LP**) problems are usually solved using the simplex method. Suppose that we want to solve the optimization problem, we would use the following syntax:

max x_1 + x_2 with constraints: $2x_1$ + x_2 ≤ 4 and x_1 + $2x_2$ ≤ 3

We assume that x_1 and x_2 are greater than or equal to 0. The first thing we need to do is convert it to the standard form. This is done by ensuring the problem is a maximization problem, that is, we convert *min z* to *max -z*. We also need to convert the inequalities to equalities by adding non-negative slack variables. The example here is already a maximization problem, so we can leave our objective function as it is. We do need to change the inequalities in the constraints to equalities:

$2x_1$ + x_2 + x_3 = 4 and x_1 + $2x_2$ + x_4 = 3

If we let *z* denote the value of the objective function, we can then write the following:

z - x_1 - x_2 = 0

We now have the following system of linear equations:

- Objective: *z - x_1 - x_2 + 0 + 0 = 0*
- Constraint 1: *$2x_1$ + x_2 + x_3 + 0 = 4*
- Constraint 2: *x_1 + $2x_2$ + 0 + x_4 = 3*

Our objective is to maximize *z*, remembering that all variables are non-negative. We can see that x_1 and x_2 appear in all the equations and are called non-basic. The x_3 and x_4 value only appear in one equation each. They are called basic variables. We can find a basic solution by assigning all non-basic variables to 0. Here, this gives us the following:

x_1 = x_2 = 0; x_3 = 4; x_4 = 3; z = 0

Is this an optimum solution, remembering that our goal is to maximize z? We can see that since z subtracts x_1 and x_2 in the first equation in our linear system, we are able to increase these variables. If the coefficients in this equation were all non-negative, then there would be no way to increase z. We will know that we have found an optimum solution when all coefficients in the objective equation are positive.

This is not the case here. So, we take one of the non-basic variables with a negative coefficient in the objective equation (say x_1, which is called the **entering variable**) and use a technique called **pivoting** to turn it from a non-basic to a basic variable. At the same time, we will change a basic variable, called the **leaving variable**, into a non-basic one. We can see that x_1 appears in both the constraint equations, so which one do we choose to pivot? Remembering that we need to keep the coefficients positive. We find that by using the pivot element that yields the lowest ratio of right-hand side of the equations to their respective entering coefficients, we can find another basic solution. For x_1, in this example, it gives us *4/2* for the first constraint and *3/1* for the second. So, we will pivot using x_1 in constraint 1.

We divide constraint *1* by 2, and get the following:

$x_1 + ½ x_2 + ½x_3 = 2$

We can now write this in terms of x_1, and substitute it into the other equations to eliminate *x1* from those equations. Once we have performed a bit of algebra, we end up with the following linear system:

$z - 1/2x_2 + 1/3 x_3 = 2$

$x_1 + 1/2 x_2 + 1/2x_3 = 2$

$3/2x_2 - 1/2x_3 + x_4 = 1$

We have another basic solution. But, is this the optimal solution? Since we still have a minus coefficient in the first equation, the answer is no. We can now go through the same pivot process with *x2*, and using the ratio rule, we find that we can pivot on *3/2x2* in the third equation. This gives us the following:
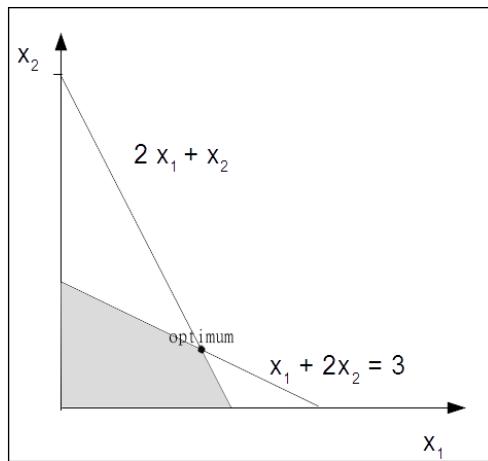
$z + 1/3x_3 + 1/3x_4 = 7/3$

$x1 + 2/3x_3 - 1/3 x_4 = 5/3$

$x2 - 1/3x_3 + 2/3 x_4 = 2/3$

This gives us the solution to $x_3 = x_4 = 0$, $x_1 = 5/3$, $x2 = 2/3$, and $z = 7/3$. This is the optimal solution because there are no more negatives in the first equation.

We can visualize this with the following graph. The shaded area is the region where we will find a feasible solution:

The two variable optimization problem

Models

Linear programming gives us a strategy for encoding real world problems into the language of computers. However, we must remember that our goal is not to just solve an instance of a problem, but to create a model that will solve unique problems from new data. This is the essence of learning. A learning model must have a mechanism to evaluate its output, and in turn, change its behavior to a state that is closer to a solution.

The model is essentially a hypothesis, that is, a proposed explanation of a phenomena. The goal is for it to apply a generalization to the problem. In the case of a supervised learning problem, knowledge gained from the training set is applied to the unlabeled test. In the case of an unsupervised learning problem, such as clustering, the system does not learn from a training set. It must learn from the characteristics of the data set itself, such as the degree of similarity. In both cases, the process is iterative. It repeats a well-defined set of tasks, which moves the model closer to a correct hypothesis.

Models are the core of a machine learning system. They are what does the learning. There are many models, with as many variations on these models, as there are unique solutions. We can see that the problems machine learning systems solve (regression, classification, association, and so on) come up in many different settings. They have been used successfully in almost all branches of science, engineering, mathematics, commerce, and also in the social sciences; they are as diverse as the domains they operate in.

This diversity of models gives machine learning systems great problem solving power. However, it can also be a bit daunting for the designer to decide which is the best model, or models, are for a particular problem. To complicate things, there are often several models that may solve your task, or your task may need several models. Which is the most accurate and efficient pathway through an original problem is something you simply cannot know when you embark upon such a project.

For our purposes here, let's break this broad canvas into three overlapping, non-mutual, and exclusive categories: geometric, probabilistic, and logical. Within these three models, a distinction must be made regarding how a model divides up the instance space. The instance space can be considered as all the possible instances of your data, regardless of whether each instance appears in the data. The actual data is a subset of the space of the instance space.

There are two approaches to dividing up this space: grouping and grading. The key difference between the two is that grouping models divide the instance space into fixed discrete units called **segments**. They have a finite resolution and cannot distinguish between classes beyond this resolution. Grading, on the other hand, forms a global model over the entire instance space, rather than dividing the space into segments. In theory, their resolution is infinite, and they can distinguish between instances no matter how similar they are. The distinction between grouping and grading is not absolute, and many models contain elements of both. For instance, a linear classifier is generally considered a grading model because it is based on a continuous function. However, there are instances that the linear model cannot distinguish between, for example, a line or surface parallel to the decision boundary.

Geometric models

Geometric models use the concept of instance space. The most obvious example of geometric models is when all the features are numerical and can become coordinates in a Cartesian coordinate system. When we only have two or three features, they are easy to visualize. However, since many machine learning problems have hundreds or thousands of features, and therefore dimensions, visualizing these spaces is impossible. However, many of the geometric concepts, such as linear transformations, still apply in this hyper space. This can help us better understand our models. For instance, we expect that many learning algorithms will be translation invariant, that is, it does not matter where we place the origin in the coordinate system. Also, we can use the geometric concept of Euclidean distance to measure any similarities between instances; this gives us a method to cluster like instances and form a decision boundary between them.

Supposing we are using our linear classifier to classify paragraphs as either happy or sad and we have devised a set of tests. Each test is associated with a weight, w, to determine how much each test contributes to the overall result.

We can simply sum up each test and multiply it by its weight to get an overall score and create a decision rule that will create a boundary, for example, if the happy score is greater than a threshold, t.

$$\sum_{i=1}^{n} w_i x_i > t$$

Each feature contributes independently to the overall result, hence the rules linearity. This contribution depends on each feature's relative weight. This weight can be positive or negative, and each individual feature is not subject to the threshold while calculating the overall score.

We can rewrite this sum with vector notation using w for a vector of weights $(w_1, w_2, ..., w_n)$ and x for a vector of test results $(x_1, x_2, ..., x_n)$. Also, if we make it an equality, we can define the decision boundary:

$w . x = t$

We can think of w as a vector pointing between the "centers of mass" of the positive (happy) examples, P, and the negative examples, N. We can calculate these centers of mass by averaging the following:

$$P = \frac{1}{n}\sum pX \quad \text{and} \quad N = \frac{1}{n}\sum nX$$

Our aim now is to create a decision boundary half way between these centers of mass. We can see that w is proportional, or equal, to *P - N*, and that *(P + N)/2* will be on the decision boundary. So, we can write the following:

$$t = (P - N) \cdot \frac{(P + N)}{2} = \frac{\left(\|P\|^2 - \|N\|^2 \right)}{2}$$
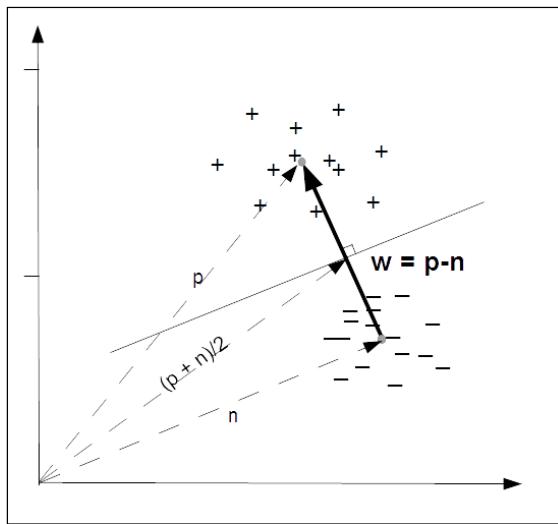
Fig of Decision boundary

In practice, real data is noisy and not necessarily that is easy to separate. Even when data is easily separable, a particular decision boundary may not have much meaning. Consider data that is sparse, such as in text classification where the number of words is large compared to the number of instances of each word. In this large area of empty instance space, it may be easy to find a decision boundary, but which is the best one? One way to choose is to use a margin to measure the distance between the decision boundary and its closest instance. We will explore these techniques later in the book.

Probabilistic models

A typical example of a probabilistic model is the Bayesian classifier, where you are given some training data (*D*), and a probability based on an initial training set (a particular hypothesis, *h*), getting the posteriori probability, *P (h/D)*.

$$P(h \mid D) = \frac{P(D \mid h) P(h)}{P(D)}$$

As an example, consider that we have a bag of marbles. We know that 40 percent of them are red and 60 percent are blue. We also know that half of the red marbles and all the blue marbles have flecks of white. When we reach into the bag to select a marble, we can feel by its texture that it has flecks. What are the chances of it being red?

Let *P(RF)* be equal to the probability that a randomly drawn marble with flecks is red:

P(FR) = the probability of a red marble with flecks is 0.5.

P(R) = the probability a marble being red is 0.4.

P(F) = the probability that a marble has flecks is *0.5 x 0.4 + 1 x 0.6= 0.8.*

$$P(R \mid F) = \frac{P(F \mid R) P(R)}{P(F)} = \frac{0.5 X 0.4}{0.8} = 0.25$$

Probabilistic models allow us to explicitly calculate probabilities, rather than just a binary true or false. As we know, the key thing we need to do is create a model that maps or features a variable to a target variable. When we take a probabilistic approach, we assume that there is an underlying random process that creates a well defined but unknown probability distribution.

Consider a spam detector. Our feature variable, *X*, could consist of a set of words that indicate the email might be spam. The target variable, *Y*, is the instance class, either spam or ham. We are interested in the conditional probability of *Y* given *X*. For each email instance, there will be a feature vector, *X*, consisting of Boolean values representing the presence of our spam words. We are trying to find out whether *Y*, our target Boolean, is representing spam or not spam.

Now, consider that we have two words, x_1 and x_2, that constitute our feature vector *X*. From our training set, we can construct a table such as the following one:

	P(Y = spam | x_1, x_2)	**P(Y = not spam | x_1, x_2)**
P(Y | x_1 = 0, x_2 = 0)	0.1	0.9
P(Y | x_1 = 0, x_2 = 1)	0.7	0.3
P(Y | x_1 = 1, x_2 = 0)	0.4	0.6
P(Y | x_1 = 1, x_2 = 1)	0.8	0.2

Table 1.1

We can see that once we begin adding more words to our feature vector, it will quickly grow unmanageable. With a feature vector of n size, we will have 2^n cases to distinguish. Fortunately, there are other methods to deal with this problem, as we shall see later.

The probabilities in the preceding table are known as posterior probabilities. These are used when we have knowledge from a prior distribution. For instance, that one in ten emails is spam. However, consider a case where we may know that X contains $x_2 = 1$, but we are unsure of the value of x_1. This instance could belong in row 2, where the probability of it being spam is **0.7**, or in row 4, where the probability is **0.8**. The solution is to average these two rows using the probability of $x_1 = 1$ in any instance. That is, the probability that a word, x_1, will appear in any email, spam or not:

$P(Y \mid x_2 = 1) = P(Y \mid x_1 = 0, x_2 = 1)P(x_1 = 0) + P(x_1 = 1, x_2 = 1)P(x_1 = 1)$

This is called a likelihood function. If we know, from a training set, that the probability that x_1 is one is 0.1 then the probability that it is zero is 0.9 since these probabilities must sum to 1. So, we can calculate the probability that an e-mail contains the spam word *0.7 * 0.9 + 0.8 * 0.1 = 0.71.*

This is an example of a likelihood function: $P(X \mid Y)$. So, why do we want to know the probability of X, which is something we all ready know, conditioned on Y, which is something we know nothing about? A way to look at this is to consider the probability of any email containing a particular random paragraph, say, the 127th paragraph of War and Peace. Clearly, this probability is small, regardless of whether the e-mail is spam or not. What we are really interested in is not the magnitude of these likelihoods, but rather their ratio. How much more likely is an email containing a particular combination of words to be spam or not spam? These are sometimes called generative models because we can sample across all the variables involved.

We can use Bayes' rule to transform between prior distributions and a likelihood function:

$$P(YX) = \frac{P(XY)P(Y)}{P(X)}$$

P(Y) is the prior probability, that is, how likely each class is, before having observed *X*. Similarly, *P(X)* is the probability without taking into account *Y*. If we have only two classes, we can work with ratios. For instance, if we want to know how much the data favors each class, we can use the following:

$$\frac{P\left(Y = spam\, X\right)}{P\left(Y = ham\, X\right)} = \frac{P\left(X\, Y = spam\right)}{P\left(X\, Y = ham\right)} \frac{P\left(Y = spam\right)}{P\left(Y = ham\right)}$$

If the odds are less than one, we assume that the class in the denominator is the most likely. If it is greater than one, then the class in the enumerator is the most likely. If we use the data from *Table 1.1*, we calculate the following posterior odds:

$$\frac{P\left(Y = spam\, x_1 = 0, x_2 = 0\right)}{P\left(Y = ham\, x_1 = 0, x_2 = 0\right)} = \frac{0.1}{0.9} = 0.11$$

$$\frac{P\left(Y = spam\, x_1 = 1, x_2 = 1\right)}{P\left(Y = ham\, x_1 = 1, x_2 = 1\right)} = \frac{0.8}{0.2} = 0.4$$

$$\frac{P\left(Y = spam\, x_1 = 0, x_2 = 1\right)}{P\left(Y = ham\, x_1 = 0, x_2 = 1\right)} = \frac{0.7}{0.3} = 2.3$$
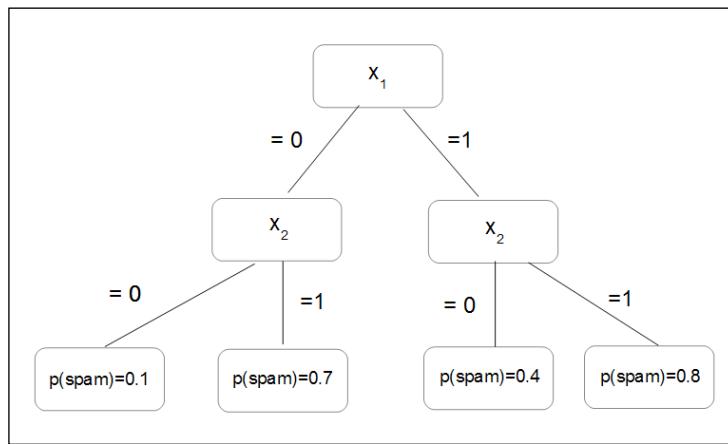
$$\frac{P\left(Y = spam\, x_1 = 1, x_2 = 0\right)}{P\left(Y = ham\, x_1 = 1, x_2 = 0\right)} = \frac{0.4}{0.6} = 0.66$$

The likelihood function is important in machine learning because it creates a generative model. If we know the probability distribution of each word in a vocabulary, together with the likelihood of each one appearing in either a spam or not spam e-mail, we can generate a random spam e-mail according to the conditional probability, *P(X | Y = spam)*.

Logical models

Logical models are based on algorithms. They can be translated into a set of formal rules that can be understood by humans. For example, if both x_1 and x_2 are 1 then the email is classified as spam.

These logical rules can be organized into a tree structure. In the following figure, we see that the instance space is iteratively partitioned at each branch. The leaves consist of rectangular areas (or hyper rectangles in the case of higher dimensions) representing segments of the instance space. Depending on the task we are solving, the leaves are labeled with a class, probability, real number, and so on.

The figure feature tree

Feature trees are very useful when representing machine learning problems; even those that, at first sight, do not appear to have a tree structure. For instance, in the Bayes classifier in the previous section, we can partition our instance space into as many regions as there are combinations of feature values. Decision tree models often employ a pruning technique to delete branches that give an incorrect result. In *Chapter 3*, *Turning Data into Information*, we will look at a number of ways to represent decision trees in Python.

> Note that decision rules may overlap and make contradictory predictions.

They are then said to be logically inconsistent. Rules can also be incomplete when they do not take into account all the coordinates in the feature space. There are a number of ways that we can address these issues, and we will look at these in detail later in the book.

Since tree learning algorithms usually work in a top down manner, the first task is to find a good feature to split on at the top of the tree. We need to find a split that will result in a higher degree of purity in subsequent nodes. By purity, I mean the degree to which training examples all belong to the same class. As we descend down the tree, at each level, we find the training examples at each node increase in purity, that is, they increasingly become separated into their own classes until we reach the leaf where all examples belong to the same class.

To look at this in another way, we are interested in lowering the entropy of subsequent nodes in our decision tree. Entropy, a measure of disorder, is high at the top of the tree (the root) and is progressively lowered at each node as the data is divided up into its respective classes.

In more complex problems, those with larger feature sets and decision rules, finding the optimum splits is sometimes not possible, at least not in an acceptable amount of time. We are really interested in creating the shallowest tree to reach our leaves in the shortest path. In the time it takes to analyze, each node grows exponentially with each additional feature, so the optimum decision tree may take longer to find than actually using a sub-optimum tree to perform the task.

An important property of logical models is that they can, to some extent, provide an explanation for their predictions. For example, consider the predictions made by a decision tree. By tracing the path from leaf to root we can determine the conditions that resulted in the final result. This is one of the advantages of logical models: they can be inspected by a human to reveal more about the problem.

Features

In the same way that decisions are only as good as the information available to us in real life, in a machine learning task, the model is only as good as its features. Mathematically, features are a function that maps from the instance space to a set of values in a particular domain. In machine learning, most measurements we make are numerical, and therefore the most common feature domain is the set of real numbers. Other common domains include Boolean, true or false, integers (say, when we are counting the occurrence of a particular feature), or finite sets such as a set of colors or shapes.

Models are defined in terms of their features. Also, single features can be turned into a model, which is known as a univariate model. We can distinguish between two uses of features. This is related to the distinction between grouping and grading.

Firstly, we can group our features by zooming into an area in the instance space. Let *f* be a feature counting the number of occurrences of a word, x_1, in an e-mail, *X*. We can set up conditions such as the following:
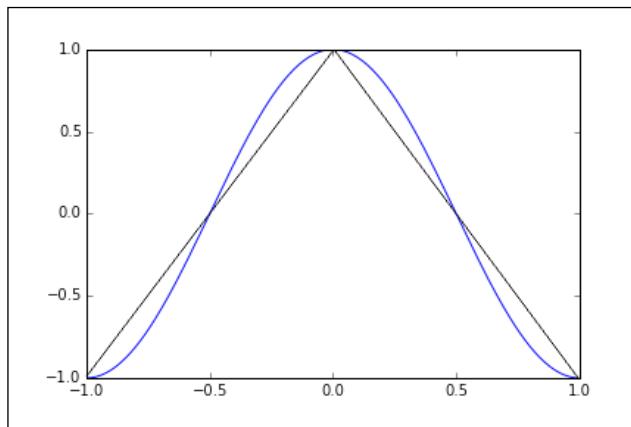
Where *f(X)=0*, representing emails that do not contain x_1 or where *f(X)>0* representing emails that contain x_1 one or more times. These conditions are called **binary splits** because they divide the instance space into two groups: those that satisfy the condition and those that don't. We can also split the instance space into more than two segments to create non-binary splits. For instance, where *f(X) = 0; 0 < F(X) < 5; F(X) > 5*, and so on.

Secondly, we can grade our features to calculate the independent contribution each one makes to the overall result. Recall our simple linear classifier, the decision rule of the following form:

$$\sum_{i=1}^{n} w_i x_i < t$$

Since this rule is linear, each feature makes an independent contribution to the score of an instance. This contribution depends on w_i. If it is positive, then a positive x_i will increase the score. If w_i is negative, a positive x_i decreases the score. If w_i is small or zero, then the contribution it makes to the overall result is negligible. It can be seen that the features make a measurable contribution to the final prediction.

These two uses of features, as splits (grouping) and predictors (grading), can be combined into one model. A typical example occurs when we want to approximate a non-linear function, say *y sin π x*, on the interval, *-1 < x < 1*. Clearly, the simple linear model will not work. Of course, the simple answer is to split the *x* axis into *-1 < x 0* and *0 <*. On each of these segments, we can find a reasonable linear approximation.

Using grouping and grading
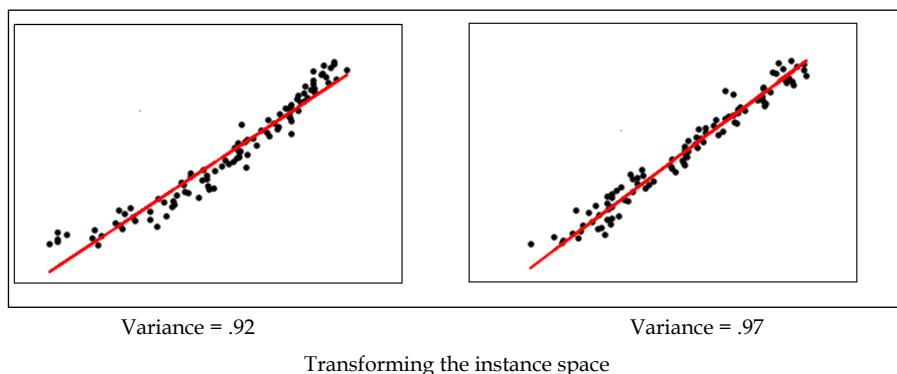
A lot of work can be done to improve our model's performance by feature construction and transformation. In most machine learning problems, the features are not necessarily explicitly available. They need to be constructed from raw datasets and then transformed into something that our model can make use of. This is especially important in problems such as text classification. In our simple spam example, we used what is known as a bag of words representation because it disregards the order of the words. However, by doing this, we lose important information about the meaning of the text.

An important part of feature construction is discretization. We can sometimes extract more information, or information that is more relevant to our task, by dividing features into relevant chunks. For instance, supposing our data consists of a list of people's precise incomes, and we are trying to determine whether there is a relationship between financial income and the suburb a person lives in. Clearly, it would be appropriate if our feature set did not consist of precise incomes but rather ranges of income, although strictly speaking, we would lose information. If we choose our intervals appropriately, we will not lose information related to our problem, and our model will perform better and give us results that are easier to interpret.

This highlights the major tasks of feature selection: separating the signal from the noise.

Real world data will invariably contain a lot of information that we do not need, as well as just plain random noise, and separating the, perhaps small, part of the data that is relevant to our needs is important to the success of our model. It is of course important that we do not throw out information that may be important to us.

Often, our features will be non-linear, and linear regression may not give us good results. A trick is to transform the instance space itself. Supposing we have data such as what is shown in the following figure. Clearly, linear regression only gives us a reasonable fit, as shown in the figure on the left-hand side. However, we can improve this result if we square the instance space, that is, we make $x = x_2$ and $y = y_2$, as shown in the figure on the right-hand side:

Variance = .92   Variance = .97

Transforming the instance space

We can go further and use a technique called the **kernel trick**. The idea is that we can create a higher dimensional implicit feature space. Pairs of data points are mapped from the raw dataset to this higher dimensional space via a specified function, sometimes called a **similarity function**.

For instance, let $x_1 = (x_1, y_1)$ and $x_2 = (x_2, y_2)$.

We create a 2D to 3D mapping, shown as follows:

$$(x, y) \Box \left(x^2, y^2, \sqrt{2}xy \right)$$

The points in the 3D space corresponding to the 2D points, x_1 and x_2, are as follows:

$$x_1' = \left(x_1^2, y_1^2, \sqrt{2}x_1 y_1 \right) \quad \text{and} \quad x_2' = \left(x_2^2, y_2^2, \sqrt{2}x_2 y_2 \right)$$

Now, the dot product of these two vectors is:

$$x_1' \bullet x_2' = x_1^2 x_2^2 + y_1^2 y_2^2 + 2x_1 y_1 x_2 y_2 = \left(x_1 x_2 + y_1 y_2 \right) = \left(x_1 \bullet x_2 \right)^2$$

We can see that by squaring the dot product in the original 2D space, we obtain the dot product in the 3D space without actually creating the feature vectors themselves. Here, we have defined the kernel $k(x_1, x_2) = (x_1, x_2)2$. Calculating the dot product in a higher dimensional space is often computationally cheaper, and as we will see, this technique is used quite widely in machine learning from **Support Vector Machines** (**SVM**), **Principle Component Analysis** (**PCA**), and correlation analysis.

The basic linear classifier we looked at earlier defines a decision boundary, $w \bullet x = t$. The vector, w, is equal to the difference between the mean of the positive example and the mean of the negative examples, $p-n$. Suppose that we have the points $n = (0,0)$ and $p = (0,1)$. Let's assume that we have obtained a positive mean from two training examples, $p1 = (-1,1)$ and $p2 = (1,1)$. Therefore, we have the following:

$$p = \frac{1}{2} \left(p_1 + p_2 \right)$$

We can now write the decision boundary as the following:

$$\frac{1}{2} p_1 \bullet x + \frac{1}{2} p_2 \bullet x - n \bullet x = t$$

Using the kernel trick, we can obtain the following decision boundary:

$$\frac{1}{2}k\left(p_1,x\right)+\frac{1}{2}k\left(p_2,x\right)-k\left(n,x\right)=t$$

With the kernel we defined earlier, we get the following:

$$k\left(p_1,x\right)=\left(-x+y\right)^2,\ k\left(p_2,x\right)=\left(x+y\right)^2\ and\ k\left(n,x\right)=0$$

We can now derive the decision boundary:

$$\frac{1}{2}\left(-x+y\right)^2+\frac{1}{2}\left(x+y\right)^2=x^2+y^2$$

This is simply a circle around the origin with a radius $\sqrt{t}$.

Using the kernel trick, on the other hand, each new instance is evaluated against each training example. In return for this more complex calculation we obtain a more flexible non-linear decision boundary.

A very interesting and important aspect is the interaction between features. One form of interaction is correlation. For example, words in a blog post, where we might perhaps expect there to be a positive correlation between the words *winter* and *cold*, and a negative correlation between *winter* and *hot*. What this means for your model depends on your task. If you are doing a sentiment analysis, you might want to consider reducing the weights of each word if they appear together since the addition of another correlated word would be expected to contribute marginally less weight to the overall result than if that word appeared by itself.

Also with regards to sentiment analysis, we often need to transform certain features to capture their meaning. For example, the phrase *not happy* contains a word that would, if we just used *1-grams*, contribute to a positive sentiment score even though its sentiment is clearly negative. A solution (apart from using *2-grams*, which may unnecessarily complicate the model) would be to recognize when these two words appear in a sequence and create a new feature, *not_happy*, with an associated sentiment score.

Selecting and optimizing features is time well spent. It can be a significant part of the design of learning systems. This iterative nature of design flips between two phases. Firstly, understanding the properties of the phenomena you are studying, and secondly, testing your ideas with experimentation. This experimentation gives us deeper insight into the phenomena, allowing us to optimize our features and gain deeper understanding, among other things, until we are satisfied about our model giving us an accurate reflection of reality.

Unified modeling language

Machine learning systems can be complex. It is often difficult for a human brain to understand all the interactions of a complete system. We need some way to abstract the system into a set of discrete functional components. This enables us to visualize our system's structure and behavior with diagrams and plots.

UML is a formalism that allows us to visualize and communicate our design ideas in a precise way. We implement our systems in code, and the underlying principles are expressed in mathematics, but there is a third aspect, which is, in a sense, perpendicular to these, and that is a visual representation of our system. The process of drawing out your design helps conceptualize it from a different perspective. Perhaps we could consider trying to triangulate a solution.

Conceptual models are theoretical devices for describing elements of a problem. They can help us clarify assumptions, prove certain properties, and give us a fundamental understanding of the structures and interactions of systems.

UML arose out of the need to both simplify this complexity and allow our designs to be communicated clearly and unambiguously to team members, clients, and other stakeholders. A model is a simplified representation of a real system. Here, we use the word *model* in a more general sense, as compared to its more precise machine learning definition. UML can be used to model almost any system imaginable. The core idea is to strip away any irrelevant and potentially confusing elements with a clear representation of core attributes and functions.

Class diagrams
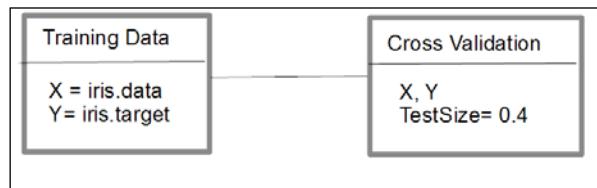
The class diagram models the static structure of a system. Classes represent abstract entities with common characteristics. They are useful because they express, and enforce, an object-oriented approach to our programming. We can see that by separating distinct objects in our code, we can work more clearly on each object as a self-contained unit. We can define it with a specific set of characteristics, and define how it relates to other objects. This enables complex programs to be broken down into separate functional components. It also allows us to subclass objects via inheritance. This is extremely useful and mirrors how we model the particularly hierarchical aspect of our world (that is, programmer is a subclass of *human*, and *Python programmer* is a subclass of programmer). Object programming can speed up the overall development time because it allows the reuse of components. There is a rich class library of developed components to draw upon. Also, the code produced tends to be easier to maintain because we can replace or change classes and are able to (usually) understand how this will affect the overall system.

In truth, object coding does tend to result in a larger code base, and this can mean that programs will be slower to run. In the end, it is not an "either, or" situation. For many simple tasks, you probably do not want to spend the time creating a class if you may never use it again. In general, if you find yourself typing the same bits of code, or creating the same type of data structures, it is probably a good idea to create a class. The big advantage of object programming is that we can encapsulate the data and the functions that operate on the data in one object. These software objects can correspond in quite a direct way with real world objects.

Designing object-oriented systems may take some time, initially. However, while establishing a workable class structure and class definitions, the coding tasks required to implement the class becomes clearer. Creating a class structure can be a very useful way to begin modeling a system. When we define a class, we are interested in a specific set of attributes, as a subset of all possible attributes or actual irrelevant attributes. It should be an accurate representation of a real system, and we need to make the judgment as to what is relevant and what is not. This is difficult because real world phenomena are complex, and the information we have about the system is always incomplete. We can only go by what we know, so our domain knowledge (the understanding of the system(s) we are trying to model), whether it be a software, natural, or human, is critically important.

Object diagrams

Object diagrams are a logical view of the system at runtime. They are a snapshot at a particular instant in time and can be understood as an instance of a class diagram. Many parameters and variables change value as the program is run, and the object diagram's function is to map these. This runtime binding is one of the key things object diagrams represent. By using links to tie objects together, we can model a particular runtime configuration. Links between objects correspond to associations between the objects class. So, the link is bound by the same constraints as the class that it enforces on its object.
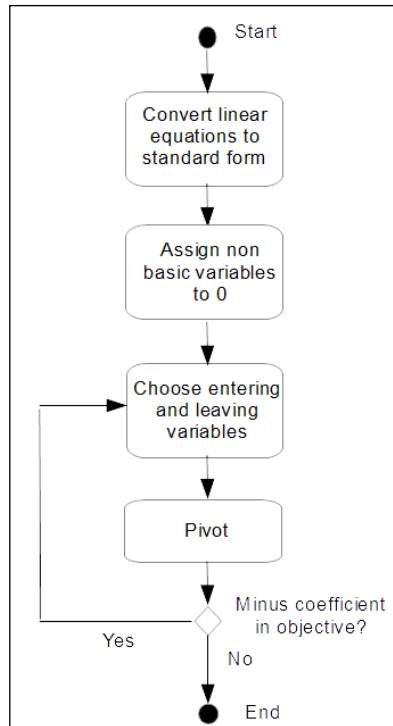
The object diagram

Both, the class diagram and the object diagram, are made of the same basic elements. While the class diagram represents an abstract blueprint of the class. The object diagram represents the real state of an object at a particular point in time. A single-object diagram cannot represent every class instance, so when drawing these diagrams, we must confine ourselves to the important instances and instances that cover the basic functionality of the system. The object diagram should clarify the association between objects and indicate the values of important variables.

Activity diagrams

The purpose of an activity diagram is to model the system's work flow by chaining together separate actions that together represent a process. They are particularly good at modeling sets of coordinated tasks. Activity diagrams are one of the most used in the UML specification because they are intuitive to understand as their formats are based on traditional flow chart diagrams. The main components of an activity diagram are actions, edges (sometimes called paths) and decisions. Actions are represented by rounded rectangles, edges are represented by arrows, and decisions are represented by a diamond. Activity diagrams usually have a start node and an end node.

A figure of an example activity diagram

State diagrams

State diagrams are used to model systems that change behavior depending on what state they are in. They are represented by states and transitions. States are represented by rounded rectangles and transitions by arrows. Each transition has a trigger, and this is written along the arrow.

Many state diagrams will include an initial pseudo state and a final state. Pseudo states are states that control the flow of traffic. Another example is the choice pseudo state. This indicates that a Boolean condition determines a transition.

A state transition system consists of four elements; they are as follows:

- $S = \{s_1, s_2, \ldots\}$: A set of states
- $A = \{a_1, a_2, \ldots\}$: A set of actions
- $E = \{e_1, e_2, \ldots\}$: A set of events
- $y: S(A \cup E) \rightarrow 2s$: A state transition function

The first element, S, is the set of all possible states the world can be in. Actions are the things an agent can do to change the world. Events can happen in the world and are not under the control of an agent. The state transition function, y, takes two things as input: a state of the world and the union of actions and events. This gives us all the possible states as a result of applying a particular action or event.
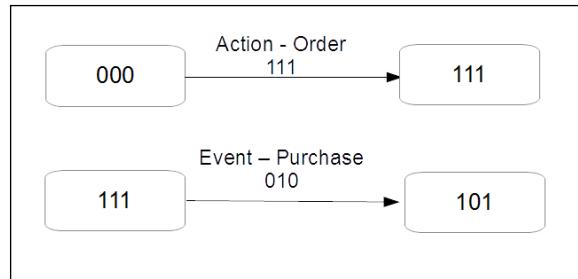
Consider that we have a warehouse that stocks three items. We consider the warehouse only stocks, at most, one of each item. We can represent the possible states of the warehouse by the following matrix:

$$S = \begin{matrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{matrix}$$

This can define similar binary matrices for E, representing the event sold, and A, which is an action order.

In this simple example, our transition function is applied to an instance (s, which is a column in S), which is $s' = s + a - e$, where s' is the system's final state, s is its initial state, and a and e are an activity and an event respectively.

We can represent this with the following transition diagram:

The figure of a transition Diagram

Summary

So far, we have introduced a broad cross-section of machine learning problems, techniques, and concepts. Hopefully by now, you have an idea of how to begin tackling a new and unique problem by breaking it up into its components. We have reviewed some of the essential mathematics and explored ways to visualize our designs. We can see that the same problem can have many different representations, and that each one may highlight different aspects. Before we can begin modeling, we need a well-defined objective, phrased as a specific, feasible, and meaningful question. We need to be clear how we can phrase the question in a way that a machine can understand.

The design process, although consisting of different and distinct activities, is not necessarily a linear process, but rather more of an iterative one. We cycle through each particular phase, proposing and testing ideas until we feel we can jump to the next phase. Sometimes we may jump back to a previous stage. We may sit at an equilibrium point, waiting for a particular event to occur; we may cycle through stages or go through several stages in parallel.

In the next chapter, we will begin our exploration of the practical tools that are available in the various Python libraries.

2
Tools and Techniques

Python comes equipped with a large library of packages for machine learning tasks.

The packages we will look at in this chapter are as follows:

- The IPython console
- NumPy, which is an extension that adds support for multi-dimensional arrays, matrices, and high-level mathematical functions
- SciPy, which is a library of scientific formulae, constants, and mathematical functions
- Matplotlib, which is for creating plots
- Scikit-learn, which is a library for machine learning tasks such as classification, regression, and clustering

There is only enough space to give you a *flavor* of these huge libraries, and an important skill is being able to find and understand the reference material for the various packages. It is impossible to present all the different functionality in a tutorial style documentation, and it is important to be able to find your way around the sometimes dense API references. A thing to remember is that the majority of these packages are put together by the open source community. They are not monolithic structures like you would expect from a commercial product, and therefore, understanding the various package taxonomies can be confusing. However, the diversity of approaches of open source software, and the fact that ideas are being contributed continually, give it an important advantage.

However, the evolving quality of open source software has its down side, especially for ML applications. For example, there was considerable reluctance on behalf of the Python machine learning user community to move from Python 2 to 3. Because Python 3 broke backwards compatibility; importantly, in terms of its numerical handling, it was not a trivial process to update the relevant packages. At the time of writing, all of the important (well important for me!) packages, and all those used in this book, were working with Python 2.7 or 3x. The major distributions of Python have Python 3 versions with a slightly different package set.

Python for machine learning

Python is a versatile general purpose programming language. It is an interpreted language and can run interactively from a console. It does not require a compiler like C++ or Java, so the development time tends to be shorter. It is available for free download and can be installed on many different operating systems including UNIX, Windows, and Macintosh. It is especially popular for scientific and mathematical applications. Python is relatively easy to learn compared to languages such as C++ and Java, with similar tasks using fewer lines of code.

Python is not the only platform for machine learning, but it is certainly one of the most used. One of its major alternatives is **R**. Like Python, it is open source, and while it is popular for applied machine learning, it lacks the large development community of Python. R is a specialized tool for machine learning and statistical analysis. Python is a general-purpose, widely-used programming language that also has excellent libraries for machine learning applications.

Another alternative is **Matlab**. Unlike R and Python, it is a commercial product. As would be expected, it contains a polished user interface and exhaustive documentation. Like R, however, it lacks the versatility of Python. Python is such an incredibly useful language that your effort to learn it, compared to the other platforms, will provide far greater pay-offs. It also has excellent libraries for network, web development, and microcontroller programming. These applications can complement or enhance your work in machine learning, all without the pain of clumsy integrations and the learning or remembering of the specifics of different languages.

IPython console

The `Ipython` package has had some significant changes with the release of version 4. A former monolithic package structure, it has been split into sub-packages. Several IPython projects have split into their own separate project. Most of the repositories have been moved to the **Jupyter** project (`jupyter.org`).

At the core of IPython is the IPython console: a powerful interactive interpreter that allows you to test your ideas in a very fast and intuitive way. Instead of having to create, save, and run a file every time you want to test a code snippet, you can simply type it into a console. A powerful feature of IPython is that it decouples the traditional read-evaluate-print loop that most computing platforms are based on. IPython puts the evaluate phase into its own process: a kernel (not to be confused with the kernel function used in machine learning algorithms). Importantly, more than one client can access the kernel. This means you can run code in a number of files and access them, for example, running a method from the console. Also, the kernel and the client do not need to be on the same machine. This has powerful implications for distributed and networked computing.

The IPython console adds command-line features, such as tab completion and `%magic` commands, which replicate terminal commands. If you are not using a distribution of Python with IPython already installed, you can start IPython by typing `ipython` into a Python command line. Typing `%quickref` into the IPython console will give you a list of commands and their function.

The IPython notebook should also be mentioned. The notebook has merged into another project known as Jupyter (`jupyter.org`). This web application is a powerful platform for numerical computing in over 40 languages. The notebook allows you to share and collaborate on live code and publish rich graphics and text.

Installing the SciPy stack

The **SciPy** stack consists of Python along with the most commonly used scientific, mathematical, and ML libraries. (visit: `scipy.org`). These include **NumPy**, **Matplotlib**, the SciPy library itself, and IPython. The packages can be installed individually on top of an existing Python installation, or as a complete distribution (**distro**). The easiest way to get started is using a distro, if you have not got Python installed on your computer. The major Python distributions are available for most platforms, and they contain everything you need in one package. Installing all the packages and their dependencies separately does take some time, but it may be an option if you already have a configured Python installation on your machine.

Most distributions give you all the tools you need, and many come with powerful developer environments. Two of the best are **Anaconda** (`www.continuum.io/downloads`) and **Canopy** (`http://www.enthought.com/products/canopy/`). Both have free and commercial versions. For reference, I will be using the Anaconda distribution of Python.

Installing the major distributions is usually a pretty painless task.

> Be aware that not all distributions include the same set of Python modules, and you may have to install modules, or reinstall the correct version of a module.

NumPY

We should know that there is a hierarchy of types for representing data in Python. At the root are immutable objects such as integers, floats, and Boolean. Built on this, we have sequence types. These are ordered sets of objects indexed by non-negative integers. They are iterative objects that include strings, lists, and tuples. Sequence types have a common set of operations such as returning an element (*s[i]*) or a slice (*s[i:j]*), and finding the length (*len(s)*) or the sum (*sum(s)*). Finally, we have mapping types. These are collections of objects indexed by another collection of key objects. Mapping objects are unordered and are indexed by numbers, strings, or other objects. The built-in Python mapping type is the dictionary.

NumPy builds on these data objects by providing two further objects: an N-dimensional array object (`ndarray`) and a universal function object (`ufunc`). The `ufunc` object provides element-by-element operations on `ndarray` objects, allowing typecasting and array broadcasting. Typecasting is the process of changing one data type into another, and broadcasting describes how arrays of different sizes are treated during arithmetic operations. There are sub-packages for linear algebra (`linalg`), random number generation (`random`), discrete Fourier transforms (`fft`), and unit testing (`testing`).

NumPy uses a `dtype` object to describe various aspects of the data. This includes types of data such as float, integer, and so on, the number of bytes in the data type (if the data is structured), and also, the names of the fields and the shape of any sub arrays. NumPy has several new data types, including the following:

- 8, 16, 32, and 64 bit `int` values
- 16, 32, and 64 bit float values
- 64 and 128 bit complex types
- `Ndarray` structured array types

We can convert between types using the `np.cast` object. This is simply a dictionary that is keyed according to destination cast type, and whose value is the appropriate function to perform the casting. Here we cast an integer to a float32:

f= np.cast['f'] (2)

NumPy arrays can be created in several ways such as converting them from other Python data structures, using the built-in array creation objects such as `arange()`, `ones()` and `zeros()`, or from files such as `.csv` or `.html`.

`Indexing` and `slicingNumPy` builds on the slicing and indexing techniques used in sequences. You should already be familiar with slicing sequences, such as lists and tuples, in Python using the `[i:j:k]` syntax, where `i` is the start index, `j` is the end, and `k` is the step. NumPy extends this concept of the selection tuple to N-dimensions.

Fire up a Python console and type the following commands:

```
import numpy as np
a=np.arange(60).reshape(3,4,5)
print(a)
```

You will observe the following:

```
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]],

       [[40, 41, 42, 43, 44],
        [45, 46, 47, 48, 49],
        [50, 51, 52, 53, 54],
        [55, 56, 57, 58, 59]]])
```

This will print the preceding 3 by 4 by 5 array. You should know that we can access each item in the array using a notation such as `a[2,3,4]`. This returns `59`. Remember that indexing begins at 0.

We can use the slicing technique to return a slice of the array.

The following image shows the `A[1:2:]` array:

```
array([[[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]]])
```

Using the ellipse (…), we can select any remaining unspecified dimensions. For example, `a[...,1]` is equivalent to `a[:,:,1]`:

```
array([[ 1,  6, 11, 16],
       [21, 26, 31, 36],
       [41, 46, 51, 56]])
```

You can also use negative numbers to count from the end of the axis:

```
In [5]: a[-1:,:,-5]
Out[5]: array([[40, 45, 50, 55]])
```

With slicing, we are creating views; the original array remains untouched, and the view retains a reference to the original array. This means that when we create a slice, even though we assign it to a new variable, if we change the original array, these changes are also reflected in the new array. The following figure demonstrates this:

```
In [6]: b=a[2,2,0:2]

In [7]: b
Out[7]: array([50, 51])

In [8]: a[2]=0 #changing a changes b

In [9]: b
Out[9]: array([0, 0])
```

Here, **a** and **b** are referring to the same array. When we assign values in `a`, this is also reflected in `b`. To copy an array rather than simply make a reference to it, we use the deep `copy()` function from the `copy` package in the standard library:

```
import copy
```

```
c=copy.deepcopy(a)
```

Here, we have created a new independent array, `c`. Any changes made in array `a` will not be reflected in array `c`.

Constructing and transforming arrays

This slicing functionality can also be used with several NumPy classes as an efficient means of constructing arrays. The `numpy.mgrid` object, for example, creates a `meshgrid` object, which provides, in certain circumstances, a more convenient alternative to `arange()`. Its primary purpose is to build a coordinate array for a specified N-dimensional volume. Refer to the following figure as an example:

```
In [10]: np.mgrid[0:4,0:4]
Out[10]:
array([[[0, 0, 0, 0],
        [1, 1, 1, 1],
        [2, 2, 2, 2],
        [3, 3, 3, 3]],

       [[0, 1, 2, 3],
        [0, 1, 2, 3],
        [0, 1, 2, 3],
        [0, 1, 2, 3]]])
```

Sometimes, we will need to manipulate our data structures in other ways. These include:

- **concatenating**: By using the `np.r_` and `np.c_` functions, we can concatenate along one or two axes using the slicing constructs. Here is an example:

```
In [11]: np.r_[-2,-1:5j,2]
Out[11]: array([-2.+0.j, -1.+0.j,  2.+0.j])
```

  Here we have used the complex number **5j** as the step size, which is interpreted by Python as the number of points, inclusive, to fit between the specified range, which here is **-1** to **1**.

- **newaxis**: This object expands the dimensions of an array:

```
In [12]: a[np.newaxis,:,:].shape
Out[12]: (1, 3, 4, 5)
```

  This creates an extra axis in the first dimension. The following creates the new axis in the second dimension:

```
In [13]: a[:,np.newaxis,:].shape
Out[13]: (3, 1, 4, 5)
```

You can also use a Boolean operator to filter:

```
a[a<5]
Out[]: array([0, 1, 2, 3, 4])
```

- Find the sum of a given axis:

```
In [14]: a.sum(2)
Out[14]:
array([[ 10,  35,  60,  85],
       [110, 135, 160, 185],
       [  0,   0,   0,   0]])
```

Here we have summed using axis 2.

Mathematical operations

As you would expect, you can perform mathematical operations such as addition, subtraction, multiplication, as well as the trigonometric functions on NumPy arrays. Arithmetic operations on different shaped arrays can be carried out by a process known as **broadcasting**. When operating on two arrays, NumPy compares their shapes element-wise from the trailing dimension. Two dimensions are compatible if they are the same size, or if one of them is 1. If these conditions are not met, then a `ValueError` exception is thrown.

This is all done in the background using the `ufunc` object. This object operates on `ndarrays` on a element-by-element basis. They are essentially wrappers that provide a consistent interface to scalar functions to allow them to work with NumPy arrays. There are over 60 `ufunc` objects covering a wide variety of operations and types. The `ufunc` objects are called automatically when you perform operations such as adding two arrays using the `+` operator.

Let's look into some additional mathematical features:

- **Vectors**: We can also create our own vectorized versions of scalar functions using the `np.vectorize()` function. It takes a Python `scalar` function or method as a parameter and returns a vectorized version of this function:

```
def myfunc(a,b):
def myfunc(a,b):
if a > b:
        return a-b
    else:
        return a + b
vfunc=np.vectorize(myfunc)
```

We will observe the following output:

```
In [18]: vfunc([1,2,3,4],[4,3,2,1])
Out[18]: array([5, 5, 1, 3])
```

- **Polynomial functions**: The `poly1d` class allows us to deal with polynomial functions in a natural way. It accepts as a parameter an array of coefficients in decreasing powers. For example, the polynomial, $2x^2 + 3x + 4$, can be entered by the following:

```
In [27]: p=np.poly1d([2,3,4])

In [28]: print(np.poly1d(p))
   2
2 x + 3 x + 4
```

We can see that it prints out the polynomial in a human-readable way. We can perform various operations on the polynomial, such as evaluating at a point:

```
In [29]: p(3)
Out[29]: 31
```

- Find the roots:

```
In [30]: p.r
Out[30]: array([-0.75+1.19895788j,
-0.75-1.19895788j])
```

We can use `asarray(p)` to give the coefficients of the polynomial an array so that it can be used in all functions that accept arrays.

As we will see, the packages that are built on NumPy give us a powerful and flexible framework for machine learning.

Matplotlib

Matplotlib, or more importantly, its sub-package `PyPlot`, is an essential tool for visualizing two-dimensional data in Python. I will only mention it briefly here because its use should become apparent as we work through the examples. It is built to work like Matlab with command style functions. Each `PyPlot` function makes some change to a `PyPlot` instance. At the core of `PyPlot` is the `plot` method. The simplest implementation is to pass plot a list or a 1D array. If only one argument is passed to plot, it assumes it is a sequence of *y* values, and it will automatically generate the *x* values. More commonly, we pass plot two 1D arrays or lists for the co-ordinates *x* and *y*. The `plot` method can also accept an argument to indicate line properties such as line width, color, and style. Here is an example:

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


x = np.arange(0., 5., 0.2)
plt.plot(x, x**4, 'r', x, x*90, 'bs', x, x**3, 'g^')
plt.show()
```

This code prints three lines in different styles: a red line, blue squares, and green triangles. Notice that we can pass more than one pair of coordinate arrays to plot multiple lines. For a full list of line styles, type the `help(plt.plot)` function.
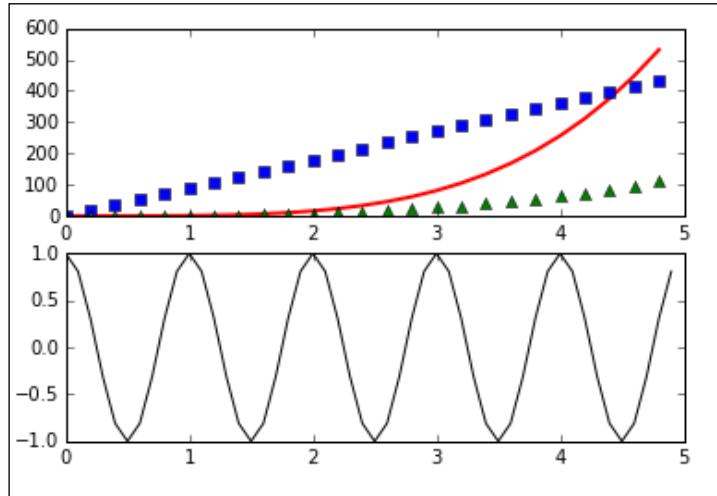
Pyplot, like Matlab, applies plotting commands to the current axes. Multiple axes can be created using the `subplot` command. Here is an example:

```python
x1 = np.arange(0., 5., 0.2)
x2 = np.arange(0., 5., 0.1)


plt.figure(1)
plt.subplot(211)
plt.plot(x1, x1**4, 'r', x1, x1*90, 'bs', x1, x1**3, 'g^',linewidth=2.0)


plt.subplot(212)
plt.plot(x2,np.cos(2*np.pi*x2), 'k')
plt.show()
```
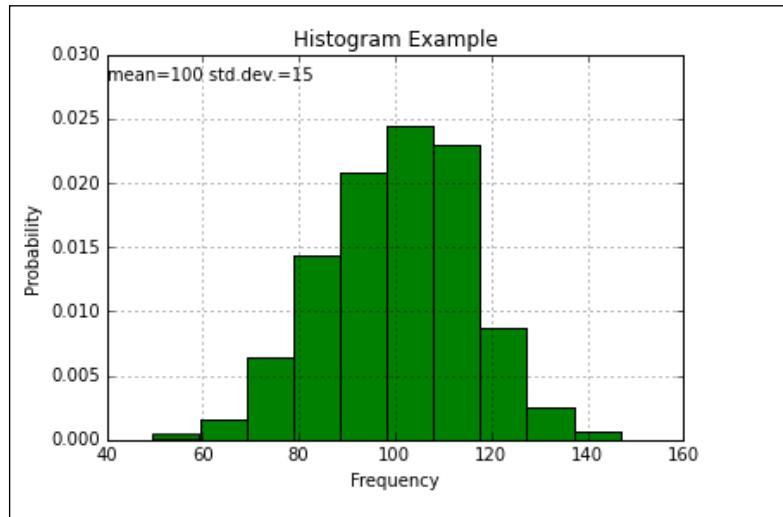
The output of the preceding code is as follows:

Another useful plot is the histogram. The `hist()` object takes an array, or a sequence of arrays, of input values. The second parameter is the number of bins. In this example, we have divided a distribution into 10 bins. The normed parameter, when set to `1` or `true`, normalizes the counts to form a probability density. Notice also that in this code, we have labeled the *x* and *y* axis, and displayed a title and some text at a location given by the coordinates:

```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(1000)
n, bins, patches = plt.hist(x, 10, normed=1, facecolor='g')
plt.xlabel('Frequency')
plt.ylabel('Probability')
plt.title('Histogram Example')
plt.text(40,.028, 'mean=100 std.dev.=15')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```
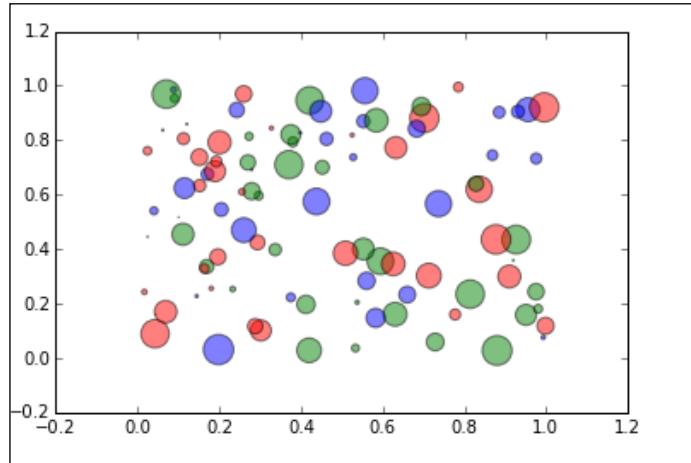
The output for this code will look like this:

The final 2D plot we are going to look at is the scatter plot. The `scatter` object takes two sequence objects, such as arrays, of the same length and optional parameters to denote color and style attributes. Let's take a look at this code:

```
N = 100
x = np.random.rand(N)
y = np.random.rand(N)
#colors = np.random.rand(N)
colors=('r','b','g')
area = np.pi * (10 * np.random.rand(N))**2  # 0 to 10 point radiuses
plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```

We will observe the following output:

Matplotlib also has a powerful toolbox for rendering 3D plots. The following code demonstrations are simple examples of 3D line, scatter, and surface plots. 3D plots are created in a very similar way to 2D plots. Here, we get the current axis with the `gca` function and set the projection parameter to 3D. All the plotting methods work much like their 2D counterparts, except that they now take a third set of input values for the *z* axis:

```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm


mpl.rcParams['legend.fontsize'] = 10


fig = plt.figure()
ax = fig.gca(projection='3d')
theta = np.linspace(-3 * np.pi, 6 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z)
```
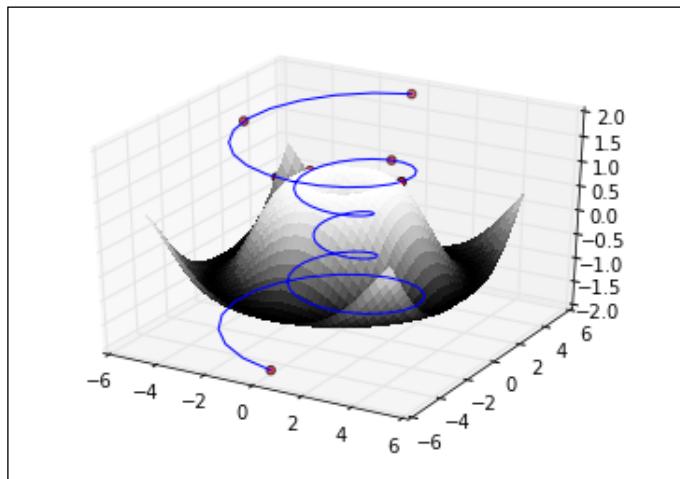
```
theta2 = np.linspace(-3 * np.pi, 6 * np.pi, 20)
z2 = np.linspace(-2, 2, 20)
r2=z2**2 +1
x2 = r2 * np.sin(theta2)
y2 = r2 * np.cos(theta2)

ax.scatter(x2,y2,z2, c= 'r')
x3 = np.arange(-5, 5, 0.25)
y3 = np.arange(-5, 5, 0.25)
x3, y3 = np.meshgrid(x3, y3)
R = np.sqrt(x3**2 + y3**2)
z3 = np.sin(R)
surf = ax.plot_surface(x3,y3,z3, rstride=1, cstride=1, cmap=cm.Greys_r,
                       linewidth=0, antialiased=False)
ax.set_zlim(-2, 2)
plt.show()
```

We will observe this output:

Pandas

The **Pandas** library builds on NumPy by introducing several useful data structures and functionalities to read and process data. Pandas is a great tool for general data munging. It easily handles common tasks such as dealing with missing data, manipulating shapes and sizes, converting between data formats and structures, and importing data from different sources.

The main data structures introduced by Pandas are:

- Series
- The DataFrame
- Panel

The DataFrame is probably the most widely used. It is a two-dimensional structure that is effectively a table created from either a NumPy array, lists, dicts, or series. You can also create a DataFrame by reading from a file.

Probably the best way to get a feel for Pandas is to go through a typical use case. Let's say that we are given the task of discovering how the daily maximum temperature has changed over time. For this example, we will be working with historical weather observations from the Hobart weather station in Tasmania. Download the following ZIP file and extract its contents into a folder called **data** in your Python working directory:

```
http://davejulian.net/mlbook/data
```

The first thing we do is create a DataFrame from it:

```
import pandas as pd
df=pd.read_csv('data/sampleData.csv')
```

Check the first few rows in this data:

```
df.head()
```

We can see that the product code and the station number are the same for each row and that this information is superfluous. Also, the days of accumulated maximum temperature are not needed for our purpose, so we will delete them as well:

```
del df['Bureau of Meteorology station number']
del df['Product code']
del df['Days of accumulation of maximum temperature']
```

Let's make our data a little easier to read by shorting the column labels:

```
df=df.rename(columns={'Maximum temperature (Degree C)':'maxtemp'})
```

We are only interested in data that is of high quality, so we include only records that have a *Y* in the quality column:

```
df=df[(df.Quality=='Y')]
```
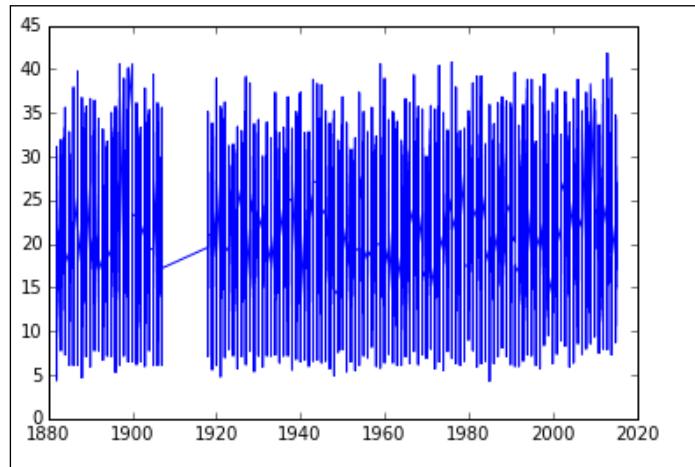
We can get a statistical summary of our data:

```
df.describe()
```

```
               Year         Month           Day        maxtemp
count  44250.000000  44250.000000  44250.000000  44250.000000
mean    1952.207503      6.536339     15.734870     16.929941
std       38.212270      3.446311      8.802089      5.030362
min     1882.000000      1.000000      1.000000      4.300000
25%     1924.000000      4.000000      8.000000     13.300000
50%     1954.000000      7.000000     16.000000     16.400000
75%     1985.000000     10.000000     23.000000     20.000000
max     2015.000000     12.000000     31.000000     41.800000
```

If we import the `matplotlib.pyplot` package, we can graph the data:

```
import matplotlib.pyplot as plt
plt.plot(df.Year, df.maxtemp)
```


Notice that PyPlot correctly formats the date axis and deals with the missing data by connecting the two known points on either side. We can convert a DataFrame into a NumPy array using the following:

```
ndarray = df.values
```

If the DataFrame contains a mixture of data types, then this function will convert them to the lowest common denominator type, which means that the one that accommodates all values will be chosen. For example, if the DataFrame consists of a mix of float16 and float32 types, then the values will be converted to float 32.

The Pandas DataFrame is a great object for viewing and manipulating simple text and numerical data. However, Pandas is probably not the right tool for more sophisticated numerical processing such as calculating the dot product, or finding the solutions to linear systems. For numerical applications, we generally use the NumPy classes.

SciPy

SciPy (pronounced sigh pi) adds a layer to NumPy that wraps common scientific and statistical applications on top of the more purely mathematical constructs of NumPy. SciPy provides higher-level functions for manipulating and visualizing data, and it is especially useful when using Python interactively. SciPy is organized into sub-packages covering different scientific computing applications. A list of the packages most relevant to ML and their functions appear as follows:

Package	Description
`cluster`	This contains two sub-packages: `cluster.vq` for K-means clustering and vector quantization. `cluster.hierachy` for hierarchical and agglomerative clustering, which is useful for distance matrices, calculating statistics on clusters, as well as visualizing clusters with dendrograms.
`constants`	These are physical and mathematical constants such as *pi* and *e*.
`integrate`	These are differential equation solvers
`interpolate`	These are interpolation functions for creating new data points within a range of known points.
`io`	This refers to input and output functions for creating string, binary, or raw data streams, and reading and writing to and from files.
`optimize`	This refers to optimizing and finding roots.
`linalg`	This refers to linear algebra routines such as basic matrix calculations, solving linear systems, finding determinants and norms, and decomposition.
`ndimage`	This is N-dimensional image processing.
`odr`	This is orthogonal distance regression.
`stats`	This refers to statistical distributions and functions.

Many of the NumPy modules have the same name and similar functionality as those in the SciPy package. For the most part, SciPy imports its NumPy equivalent and extends its functionality. However, be aware that some identically named functions in SciPy modules may have slightly different functionality compared to those in NumPy. It also should be mentioned that many of the SciPy classes have convenience wrappers in the scikit-learn package, and it is sometimes easier to use those instead.

Each of these packages requires an explicit import; here is an example:

```
import scipy.cluster
```

You can get documentation from the SciPy website (`scipy.org`) or from the console, for example, `help(sicpy.cluster)`.

As we have seen, a common task in many different ML settings is that of optimization. We looked at the mathematics of the simplex algorithm in the last chapter. Here is the implementation using SciPy. We remember simplex optimizes a set of linear equations. The problem we looked at was as follows:

Maximize $x_1 + x_2$ within the constraints of: $2x_1 + x_2 \leq 4$ and $x_1 + 2x_2 \leq 3$

The `linprog` object is probably the simplest object that will solve this problem. It is a minimization algorithm, so we reverse the sign of our objective.

From `scipy.optimize`, import `linprog`:

```
objective=[-1,-1]
con1=[[2,1],[1,2]]
con2=[4,3]
res=linprog(objective,con1,con2)
print(res)
```

You will observe the following output:

```
       nit: 2
   message: 'Optimization terminated successfully.'
    status: 0
         x: array([ 1.66666667,  0.66666667])
   success: True
       fun: -2.3333333333333335
     slack: array([ 0.,  0.])
```

There is also an `optimisation.minimize` object that is suitable for slightly more complicated problems. This object takes a solver as a parameter. There are currently about a dozen solvers available, and if you need a more specific solver, you can write your own. The most commonly used, and suitable for most problems, is the **nelder-mead** solver. This particular solver uses a **downhill simplex** algorithm that is basically a heuristic search that replaces each test point with a high error with a point located in the centroid of the remaining points. It iterates through this process until it converges on a minimum.

In this example, we use the **Rosenbrock** function as our test problem. This is a non-convex function that is often used to test optimization problems. The global minimum of this function is on a long parabolic valley, and this makes it challenging for an algorithm to find the minimum in a large, relatively flat valley. We will see more of this function:

```
import numpy as np
from scipy.optimize import minimize
def rosen(x):
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)


def nMin(funct,x0):


    return(minimize(rosen, x0, method='nelder-mead', options={'xtol':
        1e-8, 'disp': True}))


x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])


nMin(rosen,x0)
```

The output for the preceding code is as follows:

```
Optimization terminated successfully.
        Current function value: 0.000000
        Iterations: 339
        Function evaluations: 571
```

The minimize function takes two mandatory parameters. These are the objective function and the initial value of x0. The minimize function also takes an optional parameter for the solver method, in this example we use the `nelder-mead` method. The options are a solver-specific set of key-value pairs, represented as a dictionary. Here, `xtol` is the relative error acceptable for convergence, and `disp` is set to print a message. Another package that is extremely useful for machine learning applications is `scipy.linalg`. This package adds the ability to perform tasks such as inverting matrices, calculating eigenvalues, and matrix decomposition.

Scikit-learn

This includes algorithms for the most common machine learning tasks, such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing.

Scikit-learn comes with several real-world data sets for us to practice with. Let's take a look at one of these—the Iris data set:

```
from sklearn import datasets
iris = datasets.load_iris()
iris_X = iris.data
iris_y = iris.target
iris_X.shape
(150, 4)
```

The data set contains 150 samples of three types of irises (Setosa, Versicolor, and Virginica), each with four features. We can get a description on the dataset:

```
iris.DESCR
```

We can see that the four attributes, or features, are sepal width, sepal length, petal length, and petal width in centimeters. Each sample is associated with one of three classes. Setosa, Versicolor, and Virginica. These are represented by 0, 1, and 2 respectively.

Let's look at a simple classification problem using this data. We want to predict the type of iris based on its features: the length and width of its sepal and petals. Typically, scikit-learn uses estimators to implement a `fit(X, y)` method and for training a classifier, and a `predict(X)` method that if given unlabeled observations, X, returns the predicted labels, y. The `fit()` and `predict()` methods usually take a 2D array-like object.

Here, we are going to use the **K Nearest Neighbors (K-NN)** technique to solve this classification problem. The principle behind K-NN is relatively simple. We classify an unlabeled sample according to the classification of its nearest neighbors. Each data point is assigned class membership according to the majority class of a small number, *k*, of its nearest neighbors. K-NN is an example of instanced-based learning, where classification is not done according to an inbuilt model, but with reference to a labeled test set. The K-NN algorithm is known as non generalizing, since it simply remembers all its training data and compares it to each new sample. Despite, or perhaps because of, its apparent simplicity, K-NN is a very well used technique for solving a variety of classification and regression problems.

There are two different K-NN classifiers in **Sklearn**. **KNeighborsClassifier** requires the user to specify *k*, the number of nearest neighbors. **RadiusNeighborsClassifier**, on the other hand, implements learning based on the number of neighbors within a fixed radius, *r*, of each training point. KNeighborsClassifier is the more commonly used one. The optimal value for *k* is very much dependent on the data. In general, a larger *k* value is used with noisy data. The trade off being the classification boundary becomes less distinct. If the data is not uniformly sampled, then RadiusNeighborsClassifier may be a better choice. Since the number of neighbors is based on the radius, *k* will be different for each point. In sparser areas, *k* will be lower than in areas of high sample density:

```python
from sklearn.neighbors import KNeighborsClassifier as knn

from sklearn import datasets

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap


def knnDemo(X,y, n):

    #cresates the the classifier and fits it to the data
    res=0.05
    k1 = knn(n_neighbors=n,p=2,metric='minkowski')
    k1.fit(X,y)

    #sets up the grid
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, res),np.arange(x2_
min, x2_max, res))
```

```
#makes the prediction
Z = k1.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)


#creates the color map
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])


#Plots the decision surface
plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap_light)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())


#plots the samples
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)


plt.show()


iris = datasets.load_iris()
X1 = iris.data[:, 0:3:2]
X2 = iris.data[:, 0:2]
X3 = iris.data[:,1:3]
y = iris.target
knnDemo(X2,y,15)
```
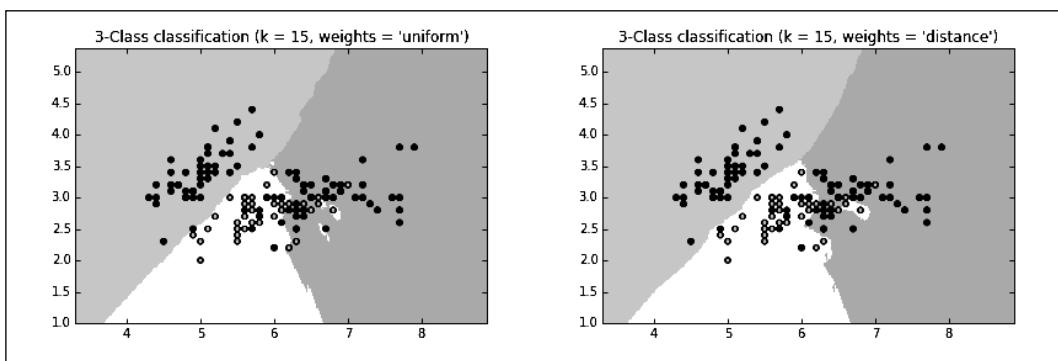
Here is the output of the preceding commands:

Let's now look at regression problems with Sklearn. The simplest solution is to minimize the sum of the squared error. This is performed by the `LinearRegression` object. This object has a `fit()` method that takes two vectors: *X*, the feature vector, and *y*, the target vector:

```
from sklearn import linear_model
clf = linear_model.LinearRegression()
clf.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
clf.coef_
array([ 0.5,  0.5])
```

The `LinearRegression` object has four optional parameters:

- `fit_intercept`: A Boolean, which if set to `false`, will assume that the data is centered, and the model will not use an intercept in its calculation. The default value is `true`.

- `normalize`: If `true`, *X* will be normalized to zero mean and unit variance before regression. This is sometimes useful because it can make interpreting the coefficients a little more explicit. The default is `false`.

- `copy_X`: Defaults to `true`. If set to `false`, it will allow *X* to be overwritten.

- `n_jobs`: Is the number of jobs to use for the computation. This defaults to `1`. This can be used to speed up computation for large problems on multiple CPUs.

Its output has the following attributes:

- `coef_`: An array of the estimated coefficients for the linear regression problem. If y is multidimensional, that is there are multiple target variables, then `coef_` will be a 2D array of the form (`n_targets`, `n_features`). If only one target variable is passed, then `coef_` will be a 1D array of length (`n_features`).

- `intercept_`: This is an array of the intercept or independent terms in the linear model.

For the **Ordinary Least Squares** to work, we assume that the features are independent. When these terms are correlated, then the matrix, *X*, can approach singularity. This means that the estimates become highly sensitive to small changes in the input data. This is known as **multicollinearity** and results in a large variance and ultimately instability. We discuss this in greater detail later, but for now, let's look at an algorithm that, to some extent, addresses these issues.

Ridge regression not only addresses the issue of multicollinearity, but also situations where the number of input variables greatly exceeds the number of samples. The `linear_model.Ridge()` object uses what is known as L2 regularization. Intuitively, we can understand this as adding a penalty on the extreme values of the weight vector. This is sometimes called **shrinkage** because it makes the average weights smaller. This tends to make the model more stable because it reduces its sensitivity to extreme values.

The Sklearn object, `linear_model.ridge`, adds a regularization parameter, `alpha`. Generally, small positive values for `alpha` improves the model's stability. It can either be a float or an array. If it is an array, it is assumed that the array corresponds to specific targets, and therefore, it must be the same size as the target. We can try this out with the following simple function:

```python
from sklearn.linear_model import Ridge
import numpy as np


def ridgeReg(alpha):

    n_samples, n_features = 10, 5
    y = np.random.randn(n_samples)
    X = np.random.randn(n_samples, n_features)
    clf = Ridge(.001)
    res=clf.fit(X, y)
    return(res)
res= ridgeReg(0.001)
print (res.coef_)
print (res.intercept_)
```

Let's now look at some scikit-learn algorithms for dimensionality reduction. This is important for machine learning because it reduces the number of input variables or features that a model has to consider. This makes a model more efficient and can make the results easier to interpret. It can also increase a model's generalization by reducing overfitting.

It is important, of course, to not discard information that will reduce the accuracy of the model. Determining what is redundant or irrelevant is the major function of dimensionality reduction algorithms. There are basically two approaches: feature extraction and feature selection. Feature selection attempts to find a subset of the original feature variables. Feature extraction, on the other hand, creates new feature variables by combining correlated variables.

Let's first look at probably the most common feature extraction algorithm, that is, **Principle Component Analysis** or **PCA**. This uses an orthogonal transformation to convert a set of correlated variables into a set of uncorrelated variables. The important information, the length of vectors, and the angle between them does not change. This information is defined in the inner product and is preserved in an orthogonal transformation. PCA constructs a feature vector in such a way that the first component accounts for as much of the variability in the data as possible. Subsequent components then account for decreasing amounts of variability. This means that, for many models, we can just choose the first few principle components until we are satisfied that they account for as much variability in our data as is required by the experimental specifications.

Probably the most versatile kernel function, and the one that gives good results in most situations, is the **Radial Basis Function** (**RBF**). The rbf kernel takes a parameter, `gamma`, which can be loosely interpreted as the inverse of the sphere of influence of each sample. A low value of gamma means that each sample has a large radius of influence on samples selected by the model. The `KernalPCA fit_transform` method takes the training vector, fits it to the model, and then transforms it into its principle components. Let's look at the commands:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import KernelPCA
from sklearn.datasets import make_circles
np.random.seed(0)
X, y = make_circles(n_samples=400, factor=.3, noise=.05)
kpca = KernelPCA(kernel='rbf', gamma=10)
X_kpca = kpca.fit_transform(X)
plt.figure()
plt.subplot(2, 2, 1, aspect='equal')
plt.title("Original space")
reds = y == 0
blues = y == 1
plt.plot(X[reds, 0], X[reds, 1], "ro")
plt.plot(X[blues, 0], X[blues, 1], "bo")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.subplot(2, 2, 3, aspect='equal')
plt.plot(X_kpca[reds, 0], X_kpca[reds, 1], "ro")
```

```
plt.plot(X_kpca[blues, 0], X_kpca[blues, 1], "bo")
plt.title("Projection by KPCA")
plt.xlabel("1st principal component in space induced by $\phi$")
plt.ylabel("2nd component")
plt.subplots_adjust(0.02, 0.10, 0.98, 0.94, 0.04, 0.35)
plt.show()
#print('gamma= %0.2f' %gamma)
```

As we have seen, a major obstacle to the success of a supervised learning algorithm is the translation from training data to test data. A labeled training set may have distinctive characteristics that are not present in new unlabeled data. We have seen that we can train our model to be quite precise on training data, yet this precision may not be translated to our unlabeled test data. Overfitting is an important problem in supervised learning and there are many techniques you can use to minimize it. A way to evaluate the estimator performance of the model on a training set is to use cross validation. Let's try this out on our iris data using a support vector machine. The first thing that we need to do is split our data into training and test sets. The train_test_split method takes two data structures: the data itself and the target. They can be either NumPy arrays, Pandas DataFrames lists, or SciPy matrices. As you would expect, the target needs to be the same length as the data. The test_size argument can either be a float between 0 and 1, representing the proportion of data included in the split, or an int representing the number of test samples. Here, we have used a test_size object as .3, indicating that we are holding out 40% of our data for testing.

In this example, we use the svm.SVC class and the .score method to return the mean accuracy of the test data in predicting the labels:

```
from sklearn.cross_validation import train_test_split
from sklearn import datasets
from sklearn import svm
from sklearn import cross_validation
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = train_test_split (iris.data,
iris.target, test_size=0.4, random_state=0)
clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
scores=cross_validation.cross_val_score(clf, X_train, y_train, cv=5)
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

You will observe the following output:

```
Accuracy: 0.99 (+/- 0.05)
```

Support vector machines have a `penalty` parameter that has to be set manually, and it is quite likely that we will run the SVC many times and adjust this parameter until we get an optimal fit. Doing this, however, leaks information from the training set to the test set, so we may still have the problem of over fitting. This is a problem for any estimator that has parameters that must be set manually, and we will explore this further in *Chapter 4*, *Models – Learning from Information*.

Summary

We have seen a basic kit of machine learning tools and a few indications of their uses on simple datasets. What you may be beginning to wonder is how these tools can be applied to real-world problems. There is considerable overlap between each of the libraries we have discussed. Many perform the same task, but add or perform the same function in a different way. Choosing which library to use for each problem is not necessarily a definitive decision. There is no best library; there is only the preferred library, and this varies from person to person, and of course, to the specifics of the application.

In the next chapter, we will look at one of the most important, and often overlooked, aspects of machine learning, that is, data.

3
Turning Data into Information

Raw data can be in many different formats and of varying quantity and quality. Sometimes, we are overwhelmed with data, and sometimes we struggle to get every last drop of information from our data. For data to become information, it requires some meaningful structure. We often have to deal with incompatible formats, inconsistencies, errors, and missing data. It is important to be able to access different parts of the dataset or extract subsets of the data based on some relational criteria. We need to spot patterns in our data and get a feel for how the data is distributed. We can use many tools to find this information hidden in data from visualizations, running algorithms, or just looking at the data in a spreadsheet.

In this chapter, we are going to introduce the following broad topics:

- Big data
- Data properties
- Data sources
- Data processing and analysis

But first, let's take a look into the following explanations:

What is data?

Data can be stored on a hard drive, streamed through a network, or captured live through sensors such as video cameras and microphones. If we are sampling from physical phenomena, such as a video or sound recording, the space is continuous and effectively infinite. Once this space is sampled, that is digitalized, a finite subset of this space has been created and at least some minimal structure has been imposed on it. The data is on a hard drive, encoded in bits, given some attributes such as a name, creation date, and so on. Beyond this, if the data is to be made use of in an application, we need to ask, "how is the data organized and what kinds of queries does it efficiently support?"

When faced with an unseen dataset, the first phase is exploration. Data exploration involves examining the components and structure of data. How many samples does it contain, and how many dimensions are in each sample? What are the data types of each dimension? We should also get a feel for the relationships between variables and how they are distributed. We need to check whether the data values are in line with what we expect. Are there are any obvious errors or gaps in the data?

Data exploration must be framed within the scope of a particular problem. Obviously, the first thing to find out is if it is likely that the dataset will provide useful answers. Is it worth our while to continue, or do we need to collect more data? Exploratory data analysis is not necessarily carried out with a particular hypothesis in mind, but perhaps with a sense of which hypotheses are likely to provide useful information.

Data is evidence that can either support or disprove a hypothesis. This evidence is only meaningful if it can be compared to a competing hypothesis. In any scientific process, we use a control. To test a hypothesis, we need to compare it to an equivalent system where the set of variables we are interested in remain fixed. We should attempt to show causality with a mechanism and explanation. We need a plausible reason for our observations. We should also consider that the real world is composed of multiple interacting components, and dealing with multivariate data can lead to exponentially increasing complexity.

It is with these things in mind, a sketch of the territory we are seeking to explore, that we approach new datasets. We have an objective, a point we hope to get to, and our data is a map through this unknown terrain.

Big data

The amount of data that's being created and stored on a global level is almost inconceivable, and it just keeps growing. Big data is a term that describes the large volume of data—both structured and unstructured. Let's now delve deeper into big data, beginning with the challenges of big data.

Challenges of big data

Big data is characterized by three challenges. They are as follows:

- The volume of the data
- The velocity of the data
- The variety of the data

Data volume

The volume problem can be approached from three different directions: **efficiency**, **scalability**, and **parallelism**. Efficiency is about minimizing the time it takes for an algorithm to process a unit of information. A component of this is the underlying processing power of the hardware. The other component, and the one that we have more control over, is ensuring that our algorithms are not wasting precious processing cycles with unnecessary tasks.

Scalability is really about brute force and throwing as much hardware at a problem as you can. Taking into account **Moore's law**, which states that the trend of computer power doubling every two years, will continue until it reaches its limit; it is clear that scalability is not, by itself, going to be able to keep up with the ever-increasing amounts of data. Simply adding more memory and faster processors is not, in many cases, going to be a cost effective solution.

Parallelism is a growing area of machine learning, and it encompasses a number of different approaches, from harnessing the capabilities of multi-core processors, to large-scale distributed computing on many different platforms. Probably, the most common method is to simply run the same algorithm on many machines, each with a different set of parameters. Another method is to decompose a learning algorithm into an adaptive sequence of queries, and have these queries processed in parallel. A common implementation of this technique is known as **MapReduce**, or its open source version, **Hadoop**.

Data velocity

The velocity problem is often approached in terms of data producers and data consumers. The rate of data transfer between the two is called the velocity, and it can be measured in interactive response times. This is the time it takes from a query being made to its response being delivered. Response times are constrained by latencies, such as hard disk read and write times, and the time it takes to transmit data across a network.

Data is being produced at ever greater rates, and this is largely driven by the rapid expansion of mobile networks and devices. The increasing instrumentation of daily life is revolutionizing the way products and services are delivered. This increasing flow of data has led to the idea of **streaming processing**. When input data is at a velocity that makes it impossible to store in its entirety, a level of analysis is necessary as the data streams, in essence, deciding what data is useful and should be stored, and what data can be thrown away. An extreme example is the **Large Hadron Collider** at CERN, where the vast majority of data is discarded. A sophisticated algorithm must scan the data as it is being generated, looking at the information needle in the data haystack. Another instance that processing data streams may be important is when an application requires an immediate response. This is becoming increasingly used in applications such as online gaming and stock market trading.
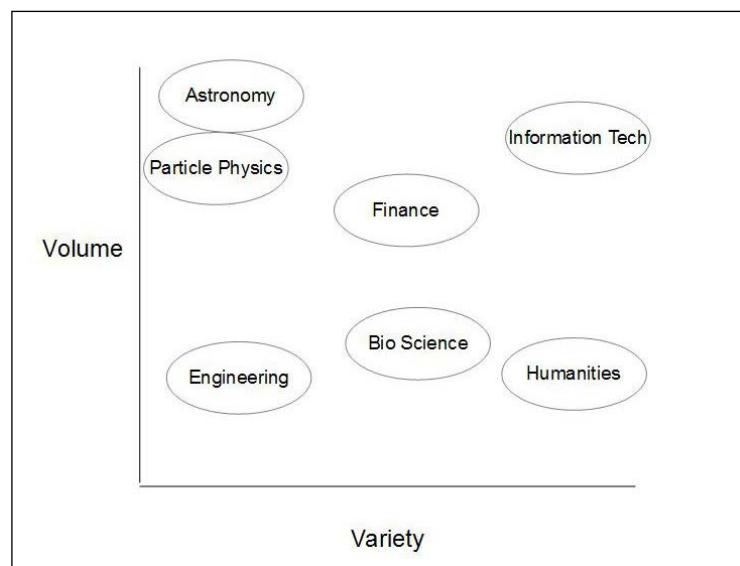
It is not just the velocity of incoming data that we are interested in; in many applications, particularly on the web, the velocity of a systems output is also important. Consider applications such as recommender systems that need to process a large amount of data and present a response in the time it takes for a web page to load.

Data variety

Collecting data from different sources invariably means dealing with misaligned data structures and incompatible formats. It also often means dealing with different semantics and having to understand a data system that may have been built on a fairly different set of logical premises. We have to remember that, very often, data is repurposed for an entirely different application from the one it was originally intended for. There is a huge variety of data formats and underlying platforms. Significant time can be spent converting data into one consistent format. Even when this is done, the data itself needs to be aligned such that each record consists of the same number of features and is measured in the same units.

Consider the relatively simple task of harvesting data from web pages. The data is already structured through the use of a mark language, typically HTML or XML, and this can help give us some initial structure. Yet, we just have to peruse the web to see that there is no standard way of presenting and tagging content in an information-relevant way. The aim of XML is to include content-relevant information in markup tags, for instance, by using tags for *author* or *subject*. However, the usage of such tags is far from universal and consistent. Furthermore, the web is a dynamic environment and many web sites go through frequent structural changes. These changes will often break web applications that expect a specific page structure.

The following diagram shows two dimensions of the big data challenge. I have included a few examples where these domains might approximately sit in this space. Astronomy, for example, has very few sources. It has a relatively small number of telescopes and observatories. Yet the volume of data that astronomers deal with is huge. On the other hand, perhaps, let's compare it to something like environmental sciences, where the data comes from a variety of sources, such as remote sensors, field surveys, validated secondary materials, and so on.

Integrating different data sets can take a significant amount of development time; up to 90 percent in some cases. Each project's data requirements will be different, and an important part of the design process is positioning our data sets with regard to these three elements.

Data models

A fundamental question for the data scientist is how the data is stored. We can talk about the hardware, and in this respect, we mean nonvolatile memory such as the hard drive of a computer or flash disk. Another way of interpreting the question (a more logical way) is how is the data organized? In a personal computer, the most visible way that data is stored is hierarchically, in nested folders and files. Data can also be stored in a table format or in a spreadsheet. When we are thinking about structure, we are interested in categories and category types, and how they are related. In a table, how many columns do we need, and in a relational data base, how are tables linked? A data model should not try to impose a structure on the data, but rather find a structure that most naturally emerges from the data.

Data models consist of three components:

- **Structure**: A table is organized into columns and rows; tree structures have nodes and edges, and dictionaries have the structure of key value pairs.

- **Constraints**: This defines the type of valid structures. For a table, this would include the fact that all rows have the same number of columns, and each column contains the same data type for every row. For example, a column, `items sold`, would only contain integer values. For hierarchical structures, a constraint would be a folder that can only have one immediate parent.

- **Operations**: This includes actions such as finding a particular value, given a key, or finding all rows where the items sold are greater than 100. This is sometimes considered separate from the data model because it is often a higher-level software layer. However, all three of these components are tightly coupled, so it makes sense to think of the operations as part of the data model.

To encapsulate raw data with a data model, we create databases. Databases solve some key problems:

- **They allow us to share data**: It gives multiple users access to the same data with varying read and write privileges.

- **They enforce a data model**: This includes not only the constraints imposed by the structure, say parent child relationships in a hierarchy, but also higher-level constraints such as only allowing one user named *bob*, or being a number between one and eight.

- **They allow us to scale**: Once the data is larger than the allocated size of our volatile memory, mechanisms are needed to both facilitate the transfer of data and also allow the efficient traversal of a large number of rows and columns.

- **Databases allow flexibility**: They essentially try to hide complexity and provide a standard way of interacting with data.

Data distributions

A key characteristic of data is its probability distribution. The most familiar distribution is the normal or Gaussian distribution. This distribution is found in many (all?) physical systems, and it underlies any random process. The normal function can be defined in terms of a **probability density function**:

$$f(x) = \frac{1}{\left(\sigma\sqrt{2}\pi\right)} \frac{e^{-(x-\mu)^2}}{\left(2\sigma^2\right)}$$
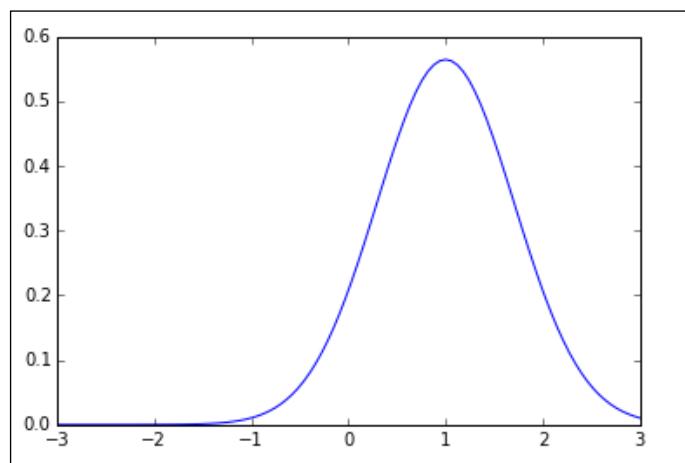
Here, **δ** (**sigma**) is the **standard deviation** and **μ** (**mu**) is the **mean**. This equation simply describes the relative likelihood a random variable, **x**, will take on a given value. We can interpret the standard deviation as the width of a bell curve, and the mean as its center. Sometimes, the term **variance** is used, and this is simply the square of the standard deviation. The standard deviation essentially measures how spread out the values are. As a general rule of thumb, in a normal distribution, 68% of the values are within 1 standard deviation of the mean, 95% of values are within 2 standard deviations of the mean, and 99.7% are within 3 standard deviations of the mean.

We can get a feel for what these terms do by running the following code and calling the `normal()` function with different values for the mean and variance. In this example, we create the plot of a normal distribution, with a mean of `1` and a variance of `0.5`:

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


def normal(mean = 0, var = 1):
    sigma = np.sqrt(var)
    x = np.linspace(-3,3,100)
    plt.plot(x,mlab.normpdf(x,mean,sigma))
    plt.show()


normal(1,0.5)
```
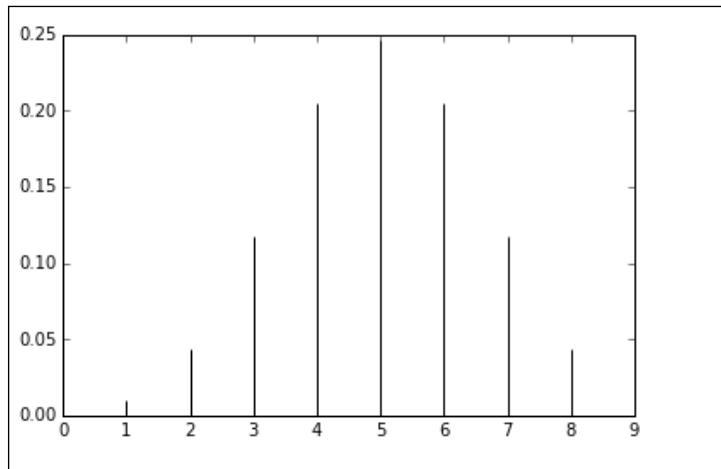
Related to the Gaussian distribution is the binomial distribution. We actually obtain a normal distribution by repeating a binomial process, such as tossing a coin. Over time, the probability approaches that half the tosses will result in heads.

$$P(x) = \frac{(n!)}{(x!(n-x)!)} p^{(x)} q^{(n-x)}$$

In this formula, **n** is the number coin tosses, **p** is the probability that half the tosses are heads, and **q** is the probability *(1-p)* that half the tosses are tails. In a typical experiment, say to determine the probability of various outcomes of a series of coin tosses, **n**, we can perform this many times, and obviously the more times we perform the experiment, the better our understanding of the statistical behavior of the system:

```python
from scipy.stats import binom
def binomial(x=10,n=10, p=0.5):
    fig, ax = plt.subplots(1, 1)
    x=range(x)
    rv = binom(n, p)
    plt.vlines(x, 0, (rv.pmf(x)), colors='k', linestyles='-')
    plt.show()
binomial()
```

You will observe the following output:

Another aspect of discrete distributions is understanding the likelihood of a given number of events occurring within a particular space and/or time. If we know that a given event occurs at an average rate, and each event occurs independently, we can describe it as a Poisson distribution. We can best understand this distribution using a probability mass function. This measures the probability of a given event that will occur at a given point in space/time.
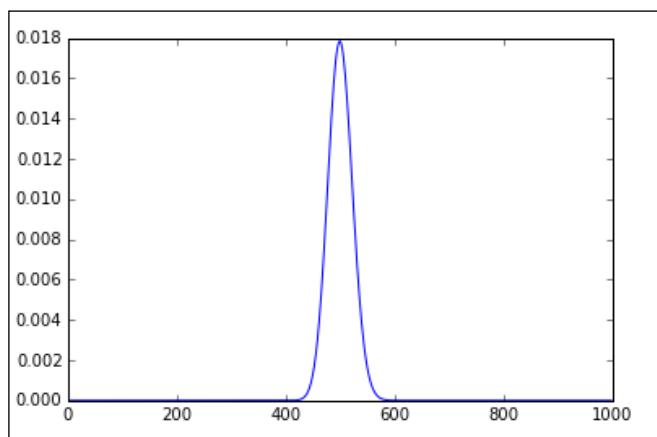
The Poisson distribution has two parameters associated with it: **lambda** ,λ, a real number greater than 0, and k, an integer that is 0, 1, 2, and so on.

$$f\left(k;\lambda\right) = \Pr\left(X = k\right) = \lambda^k \frac{e^\lambda}{k}!$$

Here, we generate the plot of a Poisson distribution using the `scipy.stats` module:

```
from scipy.stats import poisson
def pois(x=1000):
    xr=range(x)
    ps=poisson(xr)
    plt.plot(ps.pmf(x/2))
pois()
```

The output of the preceding commands is as shown in the following diagram:

We can describe continuous data distributions using probability density functions. This describes the likelihood that a continuous random variable will take on a specified value. For univariate distributions, that is, those where there is only one random variable, the probability of finding a point *X* on an interval (*a,b*) is given by the following:

$$\int_{a}^{b} f_X(x)\, dx$$

This describes the fraction of a sampled population for which a value, **x**, lies between **a** and **b**. Density functions really only have meaning when they are integrated, and this will tell us how densely a population is distributed around certain values. Intuitively, we understand this as the area under the graph of its probability function between these two points. The **Cumulative Density Function** (**CDF**) is defined as the integral of its probability density functions, **fx**:

$$F_{\gamma}(x)^{-} \int_{-m}^{x} f_X(u)\, du$$

The CDF describes the proportion of a sampled population having values for a particular variable that is less than **x**. The following code shows a discrete (binomial) cumulative distribution function. The **s1** and **s2** shape parameters determine the step size:

```
import scipy.stats as stats
def cdf(s1=50,s2=0.2):

    x = np.linspace(0,s2 * 100,s1 *2)
    cd = stats.binom.cdf
    plt.plot(x,cd(x, s1, s2))
    plt.show()
```

Data from databases

We generally interact with databases via a query language. One of the most popular query languages is MySQL. Python has a database specification, PEP 0249, which creates a consistent way to work with numerous database types. This makes the code we write more portable across databases and allows a richer span of database connectivity. To illustrate how simple this is, we are going to use the `mysql.connector` class as an example. MySQL is one of the most popular database formats, with a straight forward, human-readable query language. To practice using this class, you will need to have a MySQL server installed on your machine. This is available from `https://dev.mysql.com/downloads/mysql/`.

This should also come with a test database called **world**, which includes statistical data on world cities.

Ensure that the MySQL server is running, and run the following code:

```
import mysql.connector
from mysql.connector import errorcode


cnx = mysql.connector.connect(user='root', password='password',
                                database='world', buffered=True)
cursor=cnx.cursor(buffered=True)
query=("select * from city where population > 1000000 order by
population")
cursor.execute(query)
worldList=[]
for (city) in cursor:
    worldList.append([city[1],city[4]])
cursor.close()
cnx.close()
```

Data from the Web

Information on the web is structured into HTML or XML documents. Markup tags give us clear *hooks* for us to sample our data. Numeric data will often appear in a table, and this makes it relatively easy to use because it is already structured in a meaningful way. Let's look at a typical excerpt from an HTML document:

```
<table border="0" cellpadding="5" cellspacing="2" class="details"
width="95%">
```

```
        <tbody>

        <th>Species</th>
        <th>Data1</th>
        <th>data2</th>
        </tr>

        <td>whitefly</td>
        <td>24</td>
        <td>76</td>
        </tr>
        </tbody>
    </table>
```

This shows the first two rows of a table, with a heading and one row of data containing two values. Python has an excellent library, `Beautiful Soup`, for extracting data from HTML and XML documents. Here, we read some test data into an array, and put it into a format that would be suitable for input in a machine learning algorithm, say a linear classifier:

```
import urllib
from bs4 import BeautifulSoup
import numpy as np


url = urllib.request.urlopen
("http://interthing.org/dmls/species.html");
html = url.read()
soup = BeautifulSoup(html, "lxml")
table = soup.find("table")


headings = [th.get_text() for th in table.find("tr").find_all("th")]


datasets = []
for row in table.find_all("tr")[1:]:
    dataset = list(zip(headings, (td.get_text() for td in row.find_
all("td"))))
    datasets.append(dataset)


nd=np.array(datasets)
```

```
features=nd[:,1:,1].astype('float')
targets=(nd[:,0,1:]).astype('str')
print(features)
print(targets)
```

As we can see, this is relatively straight forward. What we need to be aware of is that we are relying on our source web page to remain unchanged, at least in terms of its overall structure. One of the major difficulties with harvesting data off the web in this way is that if the owners of the site decide to change the layout of their page, it will likely break our code.

Another data format you are likely to come across is the JSON format. Originally used for serializing Javascript objects, JSON is not, however, dependent on JavaScript. It is merely an encoding format. JSON is useful because it can represent hierarchical and multivariate data structures. It is basically a collection of key value pairs:

```
{"Languages":[{"Language":"Python","Version":"0"},{"Language":
"PHP","Version":"5"}],
"OS":{"Microsoft":"Windows 10", "Linux":"Ubuntu 14"},
"Name":"John\"the fictional\" Doe",
"location":{"Street":"Some Street", "Suburb":"Some Suburb"},
"Languages":[{"Language":"Python","Version":"0"},{"Language":"PHP"
,"Version":"5"}]
}
```

If we save the preceding JSON to a file called jsondata.json:

```
import json
from pprint import pprint


with open('jsondata.json') as file:
    data = json.load(file)


pprint(data)
```

Data from natural language

Natural language processing is one of the more difficult things to do in machine learning because it is focuses on what machines, at the moment, are not very good at: understanding the structure in complex phenomena.

As a starting point, we can make a few statements about the problem space we are considering. The number of words in any language is usually very large compared to the subset of words that are used in a particular conversation. Our data is sparse compared to the space it exists in. Moreover, words tend to appear in predefined sequences. Certain words are more likely to appear together. Sentences have a certain structure. Different social settings, such as at work, home, or out socializing; or in formal settings such as communicating with regulatory authorities, government, and bureaucratic settings, all require the use overlapping subsets of a vocabulary. A part from cues such as body language, intonation eye contact, and so forth, the social setting is probably the most important factor when trying to extract meaning from *natural* language.

To work with natural language in Python, we can use the the the **Natural Language Tool Kit** (**NLTK**). If it is not installed, you can execute the `pip install -U nltk` command.

The NLTK also comes with a large library of lexical resources. You will need to download these separately, and NLTK has a download manager accessible through the following code:

```
import nltk
```

```
nltk.download()
```

A window should open where you can browse through the various files. This includes a range of books and other written material, as well as various lexical models. To get started, you can just download the package, `Book`.

A text corpus is a large body of text consisting of numerous individual text files. NLTK comes with **corpora** from a variety of sources such as classical literature (the Gutenberg Corpus), the web and chat text, Reuter news, and corpus containing text categorized by genres such as new, editorial, religion, fiction, and so on. You can also load any collection of text files using the following code:

```
from nltk.corpus import PlaintextCorpusReader
```

```
corpusRoot= 'path/to/corpus'
```

```
yourCorpus=PlaintextCorpusReader(corpusRoot, '.*')
```

The second argument to the `PlaintextCorpusReader` method is a regular expression indicating the files to include. Here, it simply indicates that all the files in that directory are included. This second parameter could also be a list of file locations, such as `['file1', 'dir2/file2']`.

Let's take a look at one of the existing corpora, and as an example, we are going to load the Brown corpus:

```
from nltk.corpus import brown
cat=brown.categories()
print(cat)


['adventure', 'belles_lettres', 'editorial', 'fiction', 'government',
'hobbies', 'humor', 'learned', 'lore', 'mystery', 'news', 'religion',
'reviews', 'romance', 'science_fiction']
```

The Brown corpus is useful because it enables us to study the systemic differences between genres. Here is an example:

```
from nltk.corpus import brown
cats=brown.categories()
for cat in cats:
    text=brown.words(categories=cat)
    fdist = nltk.FreqDist(w.lower() for w in text)
    posmod = ['love', 'happy', 'good', 'clean']
    negmod = ['hate', 'sad', 'bad', 'dirty']
    pcount=[]
    ncount=[]
    for m in posmod:
        pcount.append(fdist[m])
    for m in negmod:
        ncount.append(fdist[m])

    print(cat + ' positive: ' + str(sum(pcount)))
    print(cat + ' negative: ' + str(sum(ncount)))
    rat=sum(pcount)/sum(ncount)
    print('ratio= %s'%rat )
    print()
```

Here, we have sort of extracted sentiment data from different genres by comparing the occurrences of four positive sentiment words with their antonyms.

Data from images

Images are a rich and easily available source of data, and they are useful for learning applications such as object recognition, grouping, grading objects, as well as image enhancement. Images, of course, can be put together as a time series. Animating images is useful for both presentation and analysis; for example, we can use video to study trajectories, monitor environments, and learn dynamic behavior.

Image data is structured as a grid or matrix with color values assigned to each pixel. We can get a feel of how this works by using the Python Image Library. For this example, you will need to execute the following lines:

```
from PIL import Image
from matplotlib import pyplot as plt
import numpy as np
image= np.array(Image.open('data/sampleImage.jpg'))
plt.imshow(image, interpolation='nearest')
plt.show()
print(image.shape)


Out[10]: (536, 800, 3)
```

We can see that this particular image is 536 pixels wide and 800 pixels high. There are 3 values per pixel, representing color values between 0 and 255, for red, green, and blue respectively. Note that the co-ordinate system's origin (*0,0*) is the top left corner. Once we have our images as NumPy arrays, we can start working with them in interesting ways, for example, taking slices:

```
im2=image[0:100,0:100,2]
```

Data from application programming interfaces

Many social networking platforms have Application programming interfaces (APIs) that give the programmer access to various features. These interfaces can generate quite large amounts of streaming data. Many of these APIs have variable support for Python 3 and some other operating systems, so be prepared to do some research regarding the compatibility of systems.

Gaining access to a platform's API usually involves registering an application with the vendor and then using supplied security credentials, such as public and private keys, to authenticate your application.

Let's take a look at the Twitter API, which is relatively easy to access and has a well-developed library for Python. To get started, we need to load the Twitter library. If you do not have it already, simply execute the `pip install twitter` command from your Python command prompt.

You will need a Twitter account. Sign in and go to `apps.twitter.com`. Click on the **Create New App** button and fill out the details on the **Create An Application** page. Once you have submitted this, you can access your credential information by clicking on your app from the application management page and then clicking on the **Keys and Access Tokens** tab.

The four items we are interested in here are the API Key, the API Secret, The Access token, and the Access Token secret. Now, to create our `Twitter` object:

```
from twitter import Twitter, OAuth
#create our twitter object
t = Twitter(auth=OAuth(accesToken, secretToken, apiKey, apiSecret))


#get our home time line
home=t.statuses.home_timeline()


#get a public timeline
anyone= t.statuses.user_timeline(screen_name="abc730")


#search for a hash tag
pycon=t.search.tweets(q="#pycon")


#The screen name of the user who wrote the first 'tweet'
user=anyone[0]['user']['screen_name']


#time tweet was created
created=anyone[0]['created_at']


#the text of the tweet
text= anyone[0]['text']
```

You will, of course, need to fill in the authorization credentials that you obtained from Twitter earlier. Remember that in a publicly accessible application, you never have these credentials in a human-readable form, and certainly not in the file itself, and preferably encrypted outside a public directory.

Signals

A form of data that is often encountered in primary scientific research is various binary streams. There are specific codecs for video and audio transmission and storage, and often, we are looking for higher-level tools to deal with each specific format. There are various signal sources we might be considering such as from a radio telescopes, sensor on a camera, or the electrical impulses from a microphone. Signals all share the same underlying principles based on wave mechanics and harmonic motion.

Signals are generally studied using time frequency analysis. The central concept here is that a continuous signal in time and space can be decomposed into frequency components. We use what is known as a **Fourier Transform** to move between the time and frequency domains. This utilizes the interesting fact that states that any given function, including non periodic functions, can be represented by a series of sine and cosine functions. This is illustrated by the following:

$$F(x) = \frac{a_0}{2} + \sum_{n=1}^{m} \left(a_n cosnx + b_n sinnx \right)$$

To make this useful, we need to find the values for a_n and b_n. We do this by multiplying both sides of the equation cosine, *mx*, and integrating. Here *m* is an integer.

$$\int_{-\pi}^{\pi} f(x) cosmx \, dx = \frac{a_0}{2} \int_{-\pi}^{\pi} cosmx \, dx + \sum a_n \int_{-\pi}^{\pi} cosnx \, cosmx \, dx + b_n \int_{-\pi}^{\pi} \sin nx \, cosmx \, dx$$

This is called an **orthogonal function,** in a similar notion to how we consider *x*, *y*, and *z* to be orthogonal in a vector space. Now, if you can remember all your trigonometric functions, you will know that *sine* times *cosine* with integer coefficients is always zero between negative *pi* and *pi*. If we do the calculation, it turns out that the middle term on the left-hand side is zero, except when *n* equals *m*. In this case, the term equals *pi*. Knowing this, we can write the following:

$$a_n = \frac{1}{\Pi} \int_{-\Pi}^{\Pi} f(x) \cos nx \, dx$$

So, in the first step, if we multiply by *sin mx* instead of *cosine mx*, then we can derive the value of b_n.

$$b_n = \Pi^1 \int_{-\Pi}^{\Pi} f(x) \sin nx \, dx$$

We can see that we have decomposed a signal into a series of *sine* values and *cosine* values. This enables us to separate the frequency components of a signal.

Data from sound

One of the most common and easy to study signals is audio. We are going to use the `soundfile` module. You can install it via `pip` if you do not have it. The `soundfile` module has a `wavfile.read` class that returns the `.wav` file data as a NumPy array. To try the following code, you will need a short 16 bit wave file called `audioSamp.wav`. This can be downloaded from `http://davejulian.net/mlbook/data`. Save it in your data directory, in your working directory:

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


sig, samplerate = sf.read('data/audioSamp.wav')
sig.shape
```
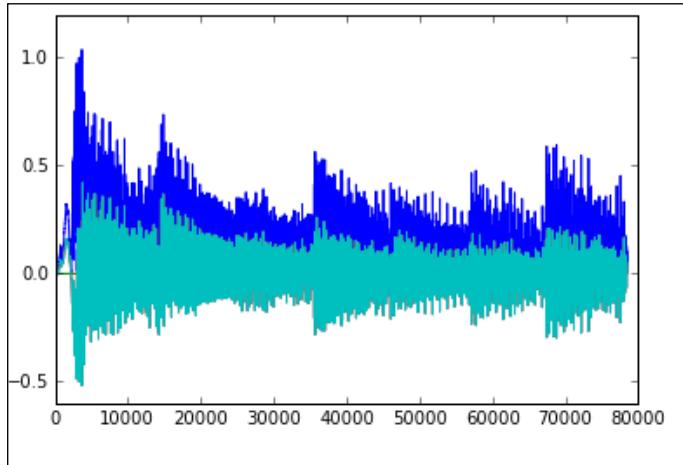
We see that the sound file is represented by a number of samples, each with two values. This is effectively the function as a vector, which describes the `.wav` file. We can, of course, create slices of our sound file:

```
slice=sig[0:500,:]
```

Here, we slice the first 500 samples. Let's calculate the Fourier transform of the slice and plot it:

```
ft=np.abs(np.fft.fft(slice))
Finally lets plot the result
plt.plot(ft)
plt.plot(slice)
```

The output of the preceding commands is as follows:

Cleaning data

To gain an understanding of which cleaning operations may be required for a particular dataset, we need to consider how the data was collected. One of the major cleaning operations involves dealing with missing data. We have already encountered an example of this in the last chapter, when we examined the temperature data. In this instance, the data had a quality parameter, so we could simply exclude the incomplete data. However, this may not be the best solution for many applications. It may be necessary to fill in the missing data. How do we decide what data to use? In the case of our temperature data, we could fill the missing values in with the average values for that time of year. Notice that we presuppose some domain knowledge, for example, the data is more or less periodic; it is in line with the seasonal cycle. So, it is a fair assumption that we could take the average for that particular date for every year we have a reliable record. However, consider that we are attempting to find a signal representing an increase in temperature due to climate change. In that case, taking the average for all years would distort the data and potentially hide a signal that could indicate warming. Once again, this requires extra knowledge and is specific about what we actually want to learn from the data.

Another consideration is that missing data may be one of three types, which are as follows:

- empty
- zero
- null

Different programming environments may treat these slightly differently. Out of the three, only zero is a measurable quantity. We know that zero can be placed on a number line before 1, 2, 3, and so on, and we can compare other numbers to zero. So, normally zero is encoded as numeric data. Empties are not necessarily numeric, and despite being empty, they may convey information. For example, if there is a field for *middle name* in a form, and the person filling out the form does not have a *middle name*, then an `empty` field accurately represents a particular situation, that is, having no middle name. Once again, this depends on the domain. In our temperature data, an `empty` field indicates missing data as it does not make sense for a particular day to have no maximum temperature. Null values, on the other hand, in computing, mean something slightly different from its everyday usage. For the computer scientist, null is not the same thing as no value or zero. Null values cannot be compared to anything else; they indicate that a field has a legitimate reason for not having an entry. Nulls are different than empty values. In our middle name example, a null value would indicate that it is unknown if the person has a middle name or not.

Another common data cleaning task is converting the data to a particular format. For our purposes here, the end data format we are interested in is a Python data structure such as a NumPy array. We have already looked at converting data from the JSON and HTML formats, and this is fairly straight forward.

Another format that we are likely to come across is the Acrobats **Portable Document Format** (**PDF**). Importing data from PDF files can be quite difficult because PDF files are built on page layout primitives, and unlike HTML or JSON, they do not have meaningful markup tags. There are several non-Python tools for turning PDFs into text such as **pdftotext**. This is a command line tool that is included in many Linux distributions and is also available for Windows. Once we have converted the PDF file into text, we still need to extract the data, and the data embedded in the document determines how we can extract it. If the data is separated from the rest of the document, say in a table, then we can use Python's text parsing tools to extract it. Alternatively, we can use a Python library for working with PDF documents such as **pdfminer3k**.

Another common cleaning task is converting between data types. There is always the risk of losing data when converting between types. This happens when the target type stores less data than the source, for instance, converting to float 16 from float 32. Sometimes, we need to convert data at the file level. This occurs when a file has an implicit typing structure, for example, a spreadsheet. This is usually done within the application that created the file. For example, an Excel spreadsheet can be saved as a comma separated text file and then imported into a Python application.

Visualizing data

There are a number of reasons for why we visually represent the data. At the data exploration stage, we can gain an immediate understanding of data properties. Visual representation serves to highlight patterns in data and suggest modeling strategies. Exploratory graphs are usually made quickly and in large numbers. We are not so much concerned with aesthetic or stylistic issues, but we simply want to see what the data looks like.
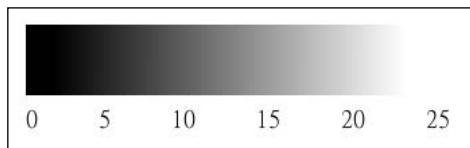
Beyond using graphs to explore data, they are a primary means of communicating information about our data. Visual representation helps clarify data properties and stimulate viewer engagement. The human visual system is the highest bandwidth channel to the brain, and visualization is the most efficient way to present a large amount of information. By creating a visualization, we can immediately get a sense of important parameters, such as the maximum, minimum, and trends that may be present in the data. Of course, this information can be extracted from data through statistical analysis, however, analysis may not reveal specific patterns in the data that visualization will. The human visual pattern recognition system is, at the moment, significantly superior to that of a machine. Unless we have clues as to what we are looking for, algorithms may not pick out important patterns that a human visual system will.

The central problem for data visualization is mapping data elements to visual attributes. We do this by first classifying the data types as nominal, ordinal, or quantitative, and then determining which visual attributes represent each data type most effectively. Nominal or categorical data refers to a name, such as the species, male or female, and so on. Nominal data does not have a specific order or numeric value. Ordinal data has an intrinsic order, such as house numbers in a street, but is different from quantitative data in that it does not imply a mathematical interval. For example, it does not make much sense to multiply or divide house numbers. Quantitative data has a numeric value such as size or volume. Clearly, certain visual attributes are inappropriate for nominal data, such as size or position; they imply ordinal or quantitative information.

Sometimes, it is not immediately clear what each data type in a particular dataset is. One way to disambiguate this is to find what operations are applicable for each data type. For example, when we are comparing nominal data, we can use equals, for instance, the species **Whitefly** is not equal to the species **Thrip**. However, we cannot use operations such as greater than or less than. It does not make sense to say, in an ordinal sense, that one species is greater than another. With ordinal data, we can apply operations such as greater than or less than. Ordinal data has an implicit order that we can map on a number line. For quantitative data, this consists of an interval, such as a date range, to which we can apply additional operations such as subtractions. For example, we can not only say that a particular date occurs after another date, but we can also calculate the difference between the two dates. With quantitative data that has a fixed axis, that is a ratio of some fixed amount as opposed to an interval, we can use operations such as division. We can say that a particular object weighs twice as much or is twice as long as another object.
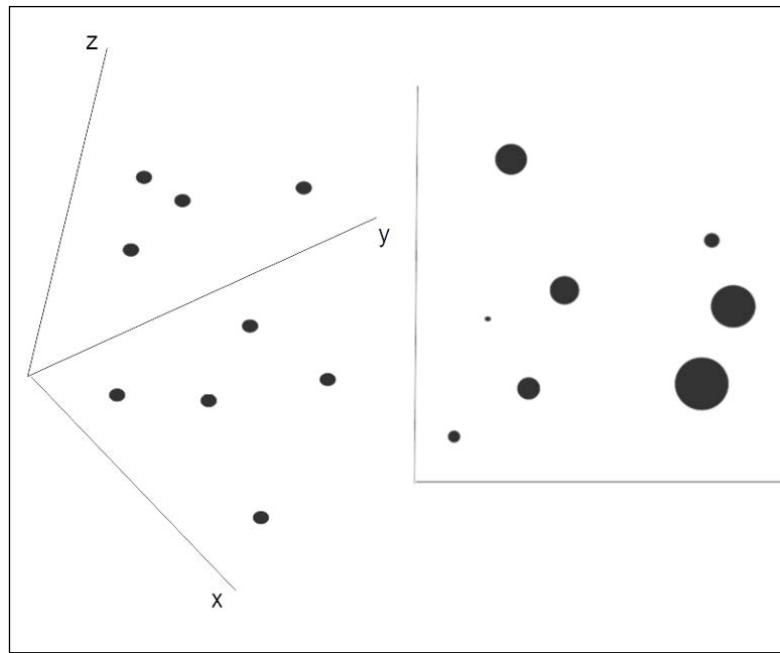
Once we are clear on our data types, we can start mapping them to attributes. Here, we will consider six visual attributes. They are position, size, texture, color, orientation, and shape. Of these, only position and size can accurately represent all three types of data. Texture, color, orientation, and shape, on the other hand, can only accurately represent nominal data. We cannot say that one shape or color is greater than another. However, we can associate a particular color or texture with a name.

Another thing to consider is the perceptual properties of these visual attributes. Research in psychology and psycho physics have established that visual attributes can be ranked in terms of how accurately they are perceived. Position is perceived most accurately, followed by length, angle, slope, area, volume, and finally, color and density, which are perceived with the least accuracy. It makes sense, therefore, to assign position and then length to the most important quantitative data. Finally, it should also be mentioned that we can encode, to some extent, ordinal data in a colors value (from dark to light) or continuous data in a color gradient. We cannot generally encode this data in a colors hue. For instance, there is no reason to perceive the color blue as somehow greater than the color red, unless you are making a reference to its frequency.

The color gradient to represent ordinal data

The next thing to consider is the number of dimensions that we need to display. For uni-variate data, that is, where we only need to display one variable, we have many choices such as dots, lines, or box plots. For bi-variate data, where we need to display two dimensions, the most common is with a scatter plot. For tri-variate data, it is possible to use a 3D plot, and this can be useful for plotting geometric functions such as manifolds. However, 3D plots have some drawbacks for many data types. It can be a problem to work out relative distances on a 3D plot. For instance, in the following figure, it is difficult to gauge the exact positions of each element. However, if we encode the *z* dimension as size, the relative values become more apparent:

Encoding Three Dimensions

There is a large design space for encoding data into visual attributes. The challenge is to find the best mapping for our particular dataset and purpose. The starting point should be to encode the most important information in the most perceptually accurate way. Effective visual coding will depict all the data and not imply anything that is not in the data. For example, length implies quantitative data, so encoding non-quantitative data into length is incorrect. Another aspect to consider is consistency. We should choose attributes that make the most sense for each data type and use consistent and well-defined visual styles.

Summary

You have learned that there are a large number of data source, formats, and structures. You have hopefully gained some understanding of how to begin working with some of them. It is important to point out that in any machine learning project, working with the data at this fundamental level can comprise a significant proportion of the overall project development time.

In the next chapter, we will look at how we can put our data to work by exploring the most common machine learning models.

4

Models – Learning from Information

So far in this book, we have examined a range of tasks and techniques. We introduced the basics of data types, structures, and properties, and we familiarized ourselves with some of the machine learning tools that are available.

In this chapter, we will look at three broad types of model:

- Logical models
- Tree models
- Rule models

The next chapter will be devoted to another important type of model—the linear model. Much of the material in this chapter is theoretical, and its purpose is to introduce some of the mathematical and logical tools needed for machine learning tasks. I encourage you to work through these ideas and formulate them in ways that may help solve problems that we come across.

Logical models

Logical models divide the instance space, that is the set of all possible or allowable, instances, into segments. The goal is to ensure that the data in each segment is homogeneous with respect to a particular task. For example, if the task is classification, then we aim to ensure that each segment contains a majority of instances of the same class.

Logical models use logical expressions to explain a particular concept. The simplest and most general logical expressions are literals, and the most common of these is equality. The equality expression can be applied to all types—nominative, numerical, and ordinal. For numerical and ordinal types, we can include the inequality literals: greater than or less than. From here, we can build more complex expressions using four logical connectives. These are conjunction (logical AND), which is denoted by ∧; disjunction (logical OR), which is denoted by ∨; implication, which is denoted by →; and negation, which is denoted by ⌐. This provides us with a way to express the following equivalences:

$$\lnot \lnot A \equiv A = A \to B \equiv \lnot A \lor B$$

$$\lnot (A \land B) \equiv \lnot A \lor \lnot B = \lnot (A \lor B) \equiv \lnot A \land \lnot B$$

We can apply these ideas in a simple example. Let's say you come across a grove of trees that all appear to be from the same species. Our goal is to identify the defining features of this tree species for use in a classification task. For simplicity sake, let's say we are just dealing with the following four features:

- Size: This has three values—small, medium, and large
- Leaf type: This has two values—scaled or non-scaled
- Fruit: This has two values—yes or no
- Buttress: This has two values—yes or no

The first tree we identify can be described by the following conjunction:

Size = Large ∧ Leaf = Scaled ∧ Fruit = No ∧ Buttress = Yes

The next tree that we come across is medium-sized. If we drop the size condition, then the statement becomes more general. That is, it will cover more samples:

Leaf = Scaled ∧ Fruit = No ∧ Buttress = Yes
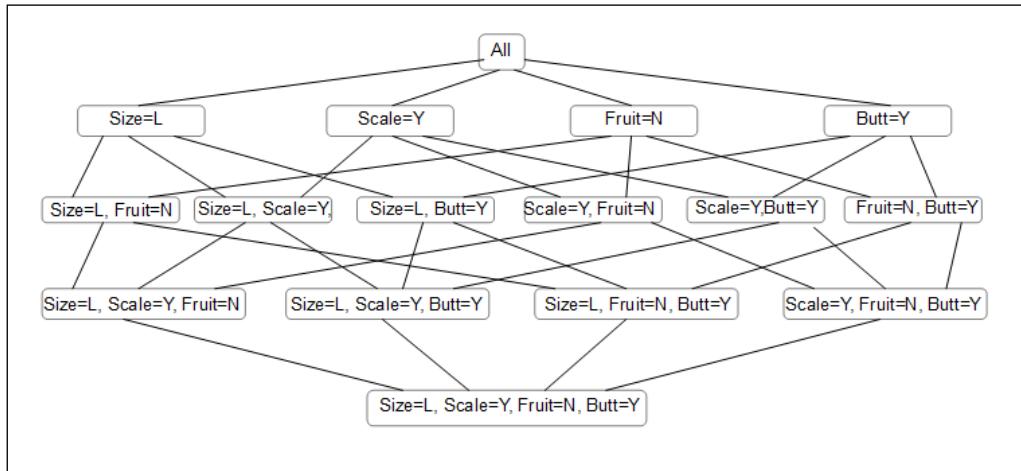
The next tree is also medium-sized, but it does not have buttresses, so we remove this condition and generalize it to the following:

Leaf = Scaled ∧ Fruit = No

The trees in the grove all satisfy this conjunction, and we conclude that they are conifers. Obviously, in a real-world example, we would use a greater range of features and values and employ more complex logical constructs. However, even in this simple example, the **instance space** is 3 2 2 2, which makes 24 possible instances. If we consider the absence of a feature as an additional value, then the **hypothesis space**, that is, the space that we can use to describe this set, is *4 3 3 3 = 108*. The number of sets of instances , or extensions, that are possible is 2^{24}. For example if you were to randomly choose a set of in. For example if you were to randomly choose a set of instances, the odds that you could find a conjunctive concept that exactly describes them is well over 100,000 to one.stances, the odds that you could find a conjunctive concept that exactly describes them is well over 100,000 to one.

Generality ordering

We can begin to map this hypothesis space from the most general statements to the most specific statements. For example, in the neighborhood of our conifer hypothesis, the space looks like this:

Here, we are ordering our hypothesis by generality. At the top is the most general hypothesis—all trees are conifers. The more general hypothesis will cover a greater number of instances, and therefore the most general hypothesis, that is, all trees are conifers, applies to all instances. Now, while this might apply to the grove we are standing in, when we attempt to apply this hypothesis to new data, that is, to trees outside the grove, it will fail. At the bottom of the preceding diagram, we have the least general hypothesis. As we make more observations and move up through the nodes, we can eliminate hypothesis and establish the next most general complete hypothesis. The most conservative generalization we can make from the data is called the **least general generalization** (**LGG**) of these instances. We can understand this as being the point in the hypothesis space where the paths upward from each of the instances intersect.

Let's describe our observations in a table:

Size	Scaled	Fruit	Buttress	Label
L	Y	N	Y	p1
M	Y	N	Y	p2
M	Y	N	N	p3
M	Y	N	Y	p4

Sooner or later, of course, you wander out of the grove and you observe negative examples—trees that are clearly not conifers. You note the following features;

Size	Scaled	Fruit	Buttress	Label
S	N	N	N	n1
M	N	N	N	n2
S	N	Y	N	n3
M	Y	N	N	n4

So, with the addition of the negative examples, we can still see that our least general complete hypothesis is still *Scale = Y ∧ Fruit = N*. However, you will notice that a negative example, *n4*, is covered. The hypothesis is therefore not consistent.

Version space

This simple example may lead you to the conclusion that there is only one LGG. But this is not necessarily true. We can expand our hypothesis space by adding a restricted form of disjunction called **internal disjunction**. In our previous example, we had three positive examples of conifers with either medium or large size. We can add a condition *Size = Medium* ∨ *Size = Large*, and we can write this as *size [m,l]*. Internal disjunction will only work with features that have more than two values because something like *Leaves = Scaled* ∨ *Leaves = Non-Scaled* is always `true`.

In the previous conifer example, we dropped the size condition to accommodate our second and third observations. This gave us the following LGG:

Leaf = Scaled ∧ *Leaf = = No*

Given our internal disjunction, we can rewrite the preceding LGG as follows:
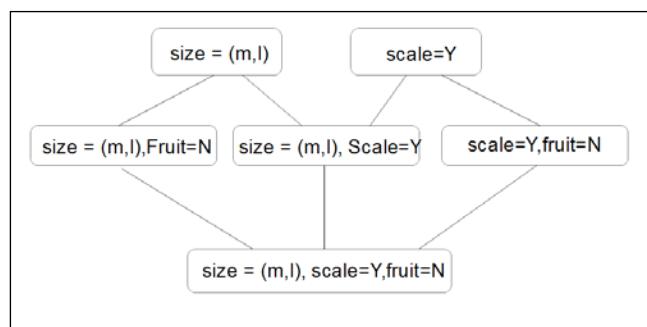
Size[m,l] ∧ *Leaf = Scaled* ∧ *Fruit = No*

Now, consider the first non-conifer, or negative non-conifer example:

Size = Small ∧ *Leaf =Non-scaled* ∧ *Fruit = No*

We can drop any of the three conditions in the LGG with the internal disjunction without covering this negative example. However, when we attempt to generalize further to single conditions, we see that *Size[m,l]* and *Leaf = Scaled* are OK but *Fruit = No* is not, since it covers the negative example.

Now, we are interested in the hypothesis that is both **complete** and **consistent**, that is, it covers all the positive examples and none of the negative. Let's now redraw our diagram considering just our four positive (*p1 - p4*) examples and one negative example (*n1*).
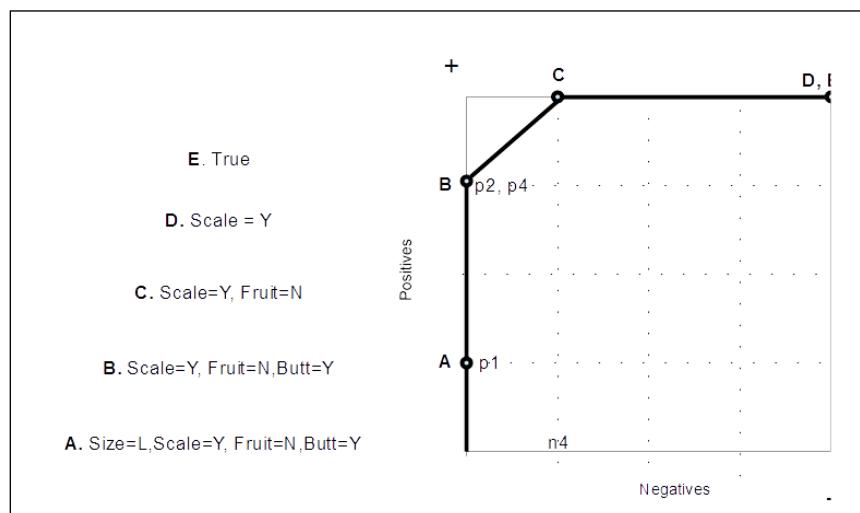
This is sometimes referred to as the **version space**. Note that we have one least general hypothesis, three intermediate, and, now, two most general hypotheses. The version space forms a convex set. This means we can interpolate between members of this set. If an element lies between a most general and least general member of the set, then it is also a member of that set. In this way, we can fully describe the version space by its most and least general members.

Consider a case where the least general generalization covers one or more of the negative instances. In such cases, we can say that the data is not **conjunctively separable** and the version space is empty. We can apply different approach whereby we search for the most general consistent hypothesis. Here we are interested in consistency as opposed to completeness. This essentially involves iterating through paths in the hypothesis space from the most general. We take downward steps by, for example, adding a conjunct or removing a value from an internal conjunct. At each step, we minimize the specialization of the resulting hypothesis.

Coverage space

When our data is not conjunctively separable, we need a way to optimize between consistency and completeness. A useful approach is in terms of mapping the **coverage space** of positive and negative instances, as shown in the following diagram:

We can see that learning a hypothesis involves finding a path through the hypothesis space ordered by generality. Logical models involve finding a pathway through a latticed structured hypothesis space. Each hypothesis in this space covers a set of instances. Each of these sets has upper and lower bounds, in and are ordered by, generality. So far, we have only used single conjunctions of literals. With a rich logical language at our disposal, why not incorporate a variety of logical connectives into our expressions? There are basically two reasons why we may want to keep our expressions simple, as follows:

- More expressive statements lead to specialization, which will result in a model overfitting training data and performing poorly on test data
- Complicated descriptions are computationally more expensive than simple descriptions

As we saw when learning about the conjunctive hypothesis, uncovered positive examples allow us to drop literals from the conjunction, making it more general. On the other hand, covered negative examples require us to increase specialization by adding literals.

Rather than describing each hypothesis in terms of conjunctions of single literals, we can describe it in terms of disjunctions of clauses, where each clause can be of the form $A \rightarrow B$. Here, A is a conjunction of literals and B is a single literal. Let's consider the following statement that covers a negative example:

Butt =Y ∧ Scaled = N ∧ Size = S ∧ ⌐ Fruit = N

To exclude this negative example, we can write the following clause:

Butt = Y ∧ Scaled = N ∧ Size = S → Fruit = N

There are of course, other clauses that exclude the negative, such as *Butt = Y → Fruit = N*; however, we are interested in the most specific clause because it is less likely to also exclude covered positives.

PAC learning and computational complexity

Given that, as we increase the complexity of our logical language, we impose a computational cost, we need a metric to gauge the *learnability* of a language. To these ends, we can use the idea of **Probably Approximately Correct** (**PAC**) learning.

When we select one hypothesis from a set of hypotheses, the goal is to ensure that our selection will have, with high probability, a low generalization error. This will perform with a high degree of accuracy on a test set. This introduces the idea of **computational complexity**. This is a formalization to gauge the computational cost of a given algorithm in relation to the accuracy of its output.

PAC learning makes allowance for mistakes on non-typical examples, and this typicality is determined by an unspecified probability distribution, *D*. We can evaluate an error rate of a hypothesis with respect to this distribution. For example, let's assume that our data is noise-free and that the learner always outputs a complete and consistent hypothesis within the training samples. Let's choose an arbitrary error rate $\epsilon < 0.5$ and a failure rate $\delta = 0.5$. We require our learning algorithm to output a hypothesis that has a probability $\geq 1 - \delta$ such that the error rate will be less than ϵ. It turns out that this will always be true for any reasonably sized training set. For example, if our hypothesis space, *H*, contains a single bad hypothesis, then the probability that it is complete and consistent on *n* independent training samples is less than or equal to $(1 - \epsilon)^n$. For any $0 \leq \epsilon \leq 1$, this probability is less than $e-n\,\epsilon$. We need to keep this below our error rate, δ, which we achieve by setting $n \geq 1/\epsilon\,ln\,1/\delta$. Now, if *H* contains a number of bad hypotheses, $k \leq |\,H\,|$, then the probability that at least one of them is complete and consistent on *n* independent samples is at maximum:

$k(1 - \epsilon)n \leq |\,H\,|\,(1 - \epsilon)n \leq |\,H\,|\,e-n\,\epsilon$

This maximum will be less than *f* if the following condition is met:

$$n \leq \frac{1}{\in} \left(\ln H + \ln \frac{1}{\delta} \right)$$

This is known as the **sample complexity** and you will notice that it is logarithmic in *1/δ* and linear in *1/ε*.

> This implies that it is exponentially cheaper to reduce the failure rate than it is to reduce the error rate.
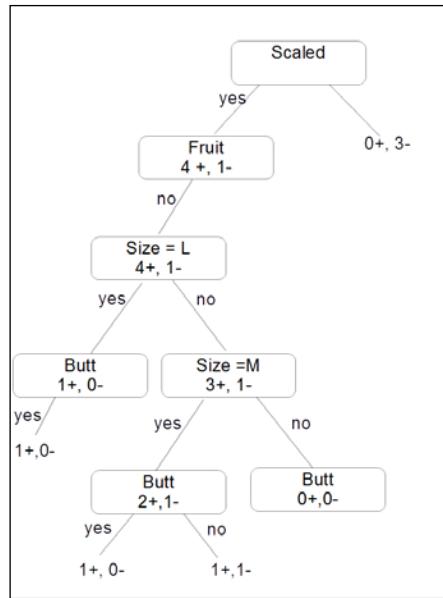
To conclude this section, I will make one further point. The hypothesis space *H* is a subset of *U*, a universe of explanation for any given phenomena. How do we know whether the correct hypothesis actually exists inside *H* rather than elsewhere in *U*? Bayes theorem shows a relationship between the relative probabilities of *H* and $\neg H$ as well as their relative prior probabilities. However, there is no real way we can know the value of $P \neg H$ because there is no way to calculate the probabilities of a hypothesis that has not yet been conceived. Moreover, the contents of this hypothesis consist of a, currently unknown, universe of possible objects. This paradox occurs in any description that uses comparative hypothesis checking where we evaluate our current hypothesis against other hypotheses within *H*. Another approach would be to find a way to evaluate *H*. We can see that, as we expand *H*, the computability of hypothesis within it becomes more difficult. To evaluate *H*, we need to restrict our universe to the universe of the known. For a human, this is a life of experiences that has been imprinted in our brains and nervous system; for a machine, it is the memory banks and algorithms. The ability to evaluate this global hypothesis space is one of the key challenges of artificial intelligence.

Tree models

Tree models are ubiquitous in machine learning. They are naturally suited to divide and conquer iterative algorithms. One of the main advantages of decision tree models is that they are naturally easy to visualize and conceptualize. They allow inspection and do not just give an answer. For example, if we have to predict a category, we can also expose the logical steps that give rise to a particular result. Also tree models generally require less data preparation than other models and can handle numerical and categorical data. On the down side, tree models can create overly complex models that do not generalize to new data very well. Another potential problem with tree models is that they can become very sensitive to changes in the input data and, as we will see later, this problem can be mitigated against using them as ensemble learners.

An important difference between decision trees and the hypothesis mapping used in the previous section is that the tree model does not use internal disjunction on features with more than two values but instead branches on each value. We can see this with the size feature in the following diagram:

Another point to note is that decision trees are more expressive than the conjunctive hypothesis and we can see this here, where we have been able to separate the data where the conjunctive hypothesis covered negative examples. This expressiveness, of course, comes with a price: the tendency to overfit on training data. A way to force generalization and reduce overfitting is to introduce an inductive bias toward less complex hypotheses.

We can quite easily implement our little example using the Sklearn `DecisionTreeClassifier` and create an image of the resultant tree:

```
from sklearn import tree

names=['size','scale','fruit','butt']
labels=[1,1,1,1,1,0,0,0]

p1=[2,1,0,1]
p2=[1,1,0,1]
p3=[1,1,0,0]
p4=[1,1,0,0]
```
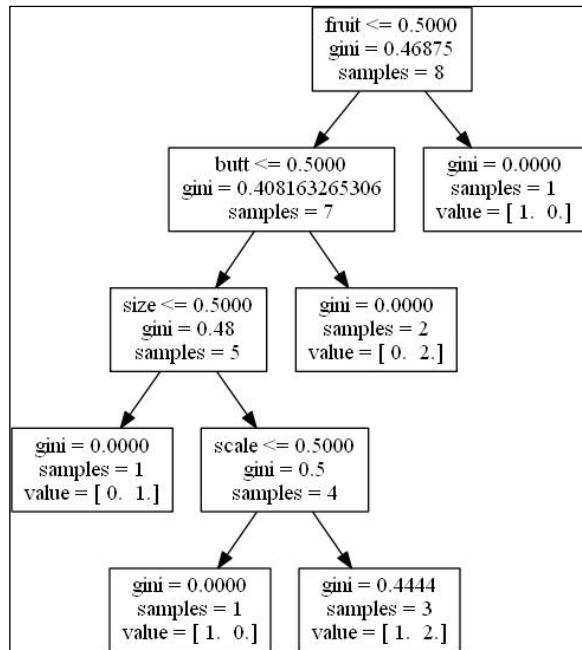
```
n1=[0,0,0,0]
n2=[1,0,0,0]
n3=[0,0,1,0]
n4=[1,1,0,0]
data=[p1,p2,p3,p4,n1,n2,n3,n4]

def pred(test, data=data):
    dtre=tree.DecisionTreeClassifier()
    dtre=dtre.fit(data,labels)
    print(dtre.predict([test]))
    with open('data/treeDemo.dot', 'w') as f:
        f=tree.export_graphviz(dtre,out_file=f,
                                  feature_names=names)
pred([1,1,0,1])
```

Running the preceding code creates a `treeDemo.dot` file. The decision tree classifier, saved as a `.dot` file, can be converted into an image file such as a `.png`, `.jpeg` or `.gif` using the **Graphiz** graph visualization software. You can download Graphviz from `http://graphviz.org/Download.php`. Once you have it installed, use it to convert the `.dot` file into an image file format of your choice.

This gives you a clear picture of how the decision tree has been split.

We can see from the full tree that we recursively split on each node, increasing the proportion of samples of the same class with each split. We continue down nodes until we reach a leaf node where we aim to have a homogeneous set of instances. This notion of purity is an important one because it determines how each node is split and it is behind the Gini values in the preceding diagram.

Purity

How do we understand the usefulness of each feature in relation to being able to split samples into classes that contain minimal or no samples from other classes? What are the indicative sets of features that give a class its label? To answer this, we need to consider the idea of purity of a split. For example, consider we have a set of Boolean instances, where D is split into $D1$ and $D2$. If we further restrict ourselves to just two classes, D^{pos} and D^{neg}, we can see that the optimum situation is where D is split perfectly into positive and negative examples. There are two possibilities for this: either where $D1^{pos} = D^{pos}$ and $D1^{neg} = \{\}$, or $D1^{neg} = D^{neg}$ and $D1^{pos} = \{\}$.

If this is true, then the children of the split are said to be pure. We can measure the impurity of a split by the relative magnitude of n^{pos} and n^{neg}. This is the empirical probability of a positive class and it can be defined by the proportion $p=n^{pos}/(n^{pos} + n^{neg})$. There are several requirements for an impurity function. First, if we switch the positive and negative class (that is, replace p with $1-p$) then the impurity should not change. Also the function should be zero when $p=0$ or $p=1$, and it should reach its maximum when $p=0.5$. In order to split each node in a meaningful way, we need an optimization function with these characteristics.
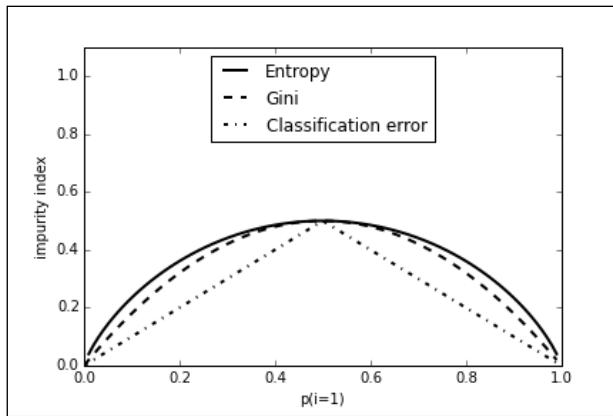
There are three functions that are typically used for impurity measures, or splitting criteria, with the following properties.

- **Minority class**: This is simply a measure of the proportion of misclassified examples assuming we label each leaf with the majority class. The higher this proportion is, the greater the number of errors and the greater the impurity of the split. This is sometimes called the **classification error**, and is calculated as *min(p,1-p)*.

- **Gini index**: This is the expected error if we label examples either positive, with probability p, or negative, with probability *1-p*. Sometimes, the square root of the Gini index is used as well, and this can have some advantages when dealing with highly skewed data where a large proportion of samples belongs to one class.

- **Entropy**: This measure of impurity is based on the expected information content of the split. Consider a message telling you about the class of a series of randomly drawn samples. The purer the set of samples, the more predictable this message becomes, and therefore the smaller the expected information. Entropy is measured by the following formula:

$$-p log_2 p - (1-p) \log_2 (1-p)$$

These three splitting criteria, for a probability range of between *0* and *1*, are plotted in the following diagram. The entropy criteria are scaled by *0.5* to enable them to be compared to the other two. We can use the output from the decision tree to see where each node lies on this curve.

Rule models

We can best understand rule models using the principles of discrete mathematics. Let's review some of these principles.

Let *X* be a set of features, the feature space, and *C* be a set of classes. We can define the ideal classifier for *X* as follows:

c: X → C

A set of examples in the feature space with class *c* is defined as follows:

$D = \{(x_1, c(x_1)), \dots , (x_n, c(x_n))\} \subseteq X \times C$

A splitting of *X* is partitioning *X* into a set of mutually exclusive subsets $X_1....X_s$, so we can say the following:

$X = X1 \cup .. \cup Xs$

This induces a splitting of *D* into $D_1...D_s$. We define *Dj* where *j = 1,...,s* and is *{(x,c(x) ∈ D | x ∈ Xj)}*.

This is just defining a subset in *X* called *Xj* where all the members of *Xj* are perfectly classified.

In the following table we define a number of measurements using sums of indicator functions. An indicator function uses the notation where *I[...]* is equal to one if the statement between the square brackets is true and zero if it is false. Here *τc(x)* is the estimate of *c(x)*.

Let's take a look at the following table:

Number of positives	$$P = \sum_{(x \in D)} I\left[c(x) = pos\right]$$
Number of negatives	$$N = \sum_{(x \in D)} I\left[c(x) = neg\right]$$
True positives	$$TP = \sum_{(x \in D)} I\left[\tau c(x) = c(x) = pos\right]$$
True negatives	$$TN = \sum_{(x \in D)} I\left[\tau c(x) = c(x) = neg\right]$$
False positives	$$FP = \sum_{(x \in D)} I\left[\tau c(x) = pos, c(x) = neg\right]$$
False negatives	$$FN = \sum_{(x \in D)} I\left[\tau(x) = neg, c(x) = pos\right]$$
Accuracy	$$acc = \frac{1}{D}\sum_{(x \in D)} I\left[\tau c(x) = c(x)\right]$$
Error rate	$$err = \frac{1}{D}\sum_{(x \in D)} I\left[\tau c(x) \neq c(x)\right]$$
True positive rate (sensitivity, recall)	$$tpr = \frac{\left(\sum_{(x \in D)} I\left[\tau c(x) = c(x) = pos\right]\right)}{\left(\sum_{(x \in D)} I\left[c(x) = pos\right]\right)} = \frac{TP}{P}$$
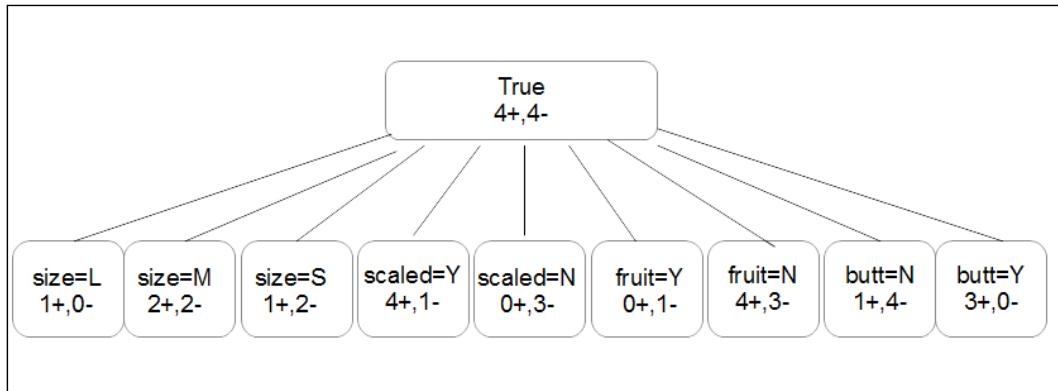
True negative rate (negative recall)	$tnr = \dfrac{\left(\sum_{(x \in D)} I\left[\tau c(x) = c(x) = neg\right]\right)}{\left(\sum_{(x \in D)} I\left[c(x) = neg\right]\right)} = \dfrac{TN}{N}$
Precision, confidence	$prec = \dfrac{\left(\sum_{(x \in D)} I\left[\tau c(x) = c(x) = pos\right]\right)}{\left(\sum_{(x \in D)} I\left[\tau c(x) = pos\right]\right)} = \dfrac{TP}{(TP + FP)}$

Rule models comprise not only sets or lists of rules, but importantly, a specification on how to combine these rules to form predictions. They are a logical model but differ from the tree approach in that, trees split into mutually exclusive branches, whereas rules can overlap, possibly carrying additional information. In supervised learning there are essentially two approaches to rule models. One is to find a combination of literals, as we did previously, to form a hypothesis that covers a sufficiently homogeneous set of samples, and then find a label. Alternatively, we can do the opposite; that is, we can first select a class and then find rules that cover a sufficiently large subset of samples of that class. The first approach tends to lead to an ordered list of rules, and in the second approach, rules are an unordered set. Each deals with overlapping rules in its own characteristic way, as we will see. Let's look at the ordered list approach first.

The ordered list approach

As we add literals to a conjunctive rule, we aim to increase the homogeneity of each subsequent set of instances covered by the rule. This is similar to constructing a path in the hypothesis space as we did for our logical trees in the last section. A key difference with the rule approach is that we are only interested in the purity of one of the children, the one where the added literal is true. With tree-based models, we use the weighted average of both children to find the purity of both branches of a binary split. Here, we are still interested in calculating the purity of subsequent rules; however, we only follow one side of each split. We can still use the same methods for finding purity, but we no longer need to average over all children. As opposed to the divide and conquer strategy of decision trees, rule-based learning is often described as separate and conquer.

Let's briefly consider an example using our conifer categorization problem from the previous section.

There are several options for choosing a rule that will result in the purest split. Supposing we choose the rule *If scaled = N then class is negative*, we have covered three out of four negative samples. In the next iteration, we remove these samples from consideration and continue this process of searching for literals with maximum purity. Effectively, what we are doing is building an ordered list of rules joined with the `if` and `else` clauses. We can rewrite our rules to be mutually exclusive, and this would mean that the set of rules does not need to be ordered. The tradeoff here is that we would have to use either negated literals or internal disjunctions to deal with features that have more than two values.

There are certain refinements we can make to this model. For example, we can introduce a stopping criterion that halts iteration if certain conditions are met, such as in the case of noisy data where we may want to stop iteration when the number of samples in each class falls below a certain number.

Ordered rule models have a lot in common with decision trees, especially, in that, they use an objective function based on the notion of purity that is the relative number of positive and negative class instances in each split. They have structures that are easy to visualize and they are used in many different machine learning settings.

Set-based rule models

With set based rule models rules are learned one class at a time, and our objective function simply becomes maximize *p*, rather than minimizing min(*p,1-p*). Algorithms that use this method typically iterate over each class and only cover samples of each class that are removed after a rule is found. Set-based models use precision (refer to table 4-1) as a search heuristic and this can make the model focus too much on the purity of the rule; it may miss near pure rules that can be further specialized to form a pure rule. Another approach, called **beam search**, uses a heuristic to order a predetermined number of best partial solutions.

> Ordered lists give us a convex coverage for the training set. This is not necessarily true of the uncorded set-based approach where there is no global optimum order for a given set of rules. Because of this, we have access to rule overlaps expressed as a conjunction *A∧B*, where *A* and *B* are two rule sets. If these two rules are in an ordered list, we have either, if the order is *AB*, *A = (A∧B) ∨ (A∧ ⌐B)* or, if the order is *BA*, *B = (A∧B) ∨ (⌐A∧B)*. This means that the rule space is potentially enlarged; however, because we have to estimate the coverage of overlaps, we sacrifice convexity.

Rule models, in general, are well suited to predictive models. We can, as we will see later, extend our rule models to perform such tasks as clustering and regression. Another important application of rule models is to build **descriptive models**. When we are building classification models, we generally look for rules that will create pure subsets of the training samples. However, this not necessarily true if we are looking for other distinguishing characteristics of a particular sample set. This is sometimes referred to as **subgroup discovery**. Here, we are not interested in a heuristic that is based on class purity but rather in one that looks for distinguishing class distributions. This is done using a defined quality function based on the idea of local exceptional testing. This function can take the form *q=TP/(FP +g)*. Here *g* is a generalization factor that determines the allowable number of nontarget class instances relative to the number of instances covered by the rule. For a small value of *g*, say less than *1*, rules will be generated that are more specific because every additional nontarget example incurs greater relative *expense*. Higher values of *g*, say greater than *10*, create more general rules covering more nontarget samples. There is no theoretical maximum value for *g*; however, it does not make much sense for it to exceed the number of samples. The value of *g* is governed by the size of the data and the proportion of positive samples. The value of *g* can be varied, thus guiding subgroup discovery to certain points in the *TP* versus *FP* space.

We can use subjective or objective quality functions. We can incorporate subjective *interestingness* coefficients into the model to reflect things such as understandability, unexpectedness, or, based on templates describing the interesting class, relationship patterns. Objective measurements are derived from the statistical and structural properties of the data itself. They are very amenable to the use of coverage plots to highlight subgroups that have statistical properties that differ from the population as a whole.
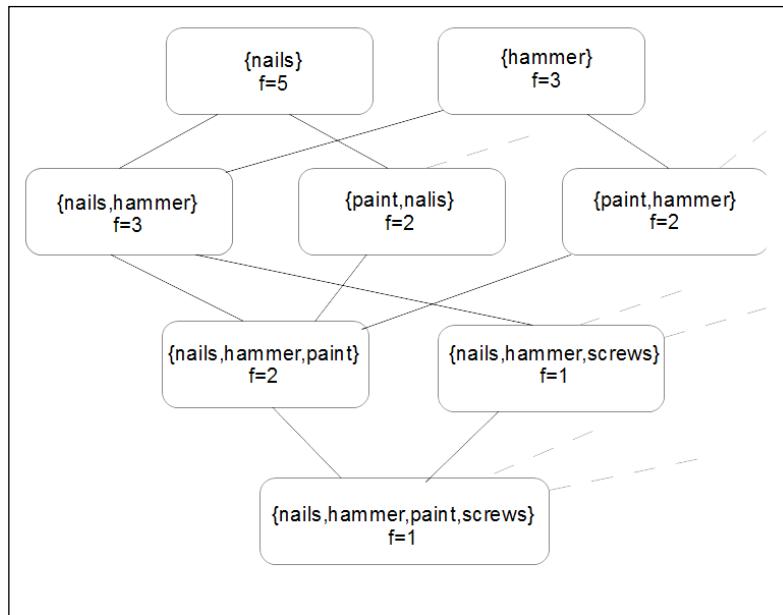
Finally, in this section on rule-based models, we will consider rules that can be learned entirely unsupervised. This is called **association rule learning**, and its typical use cases include data mining, recommender systems and natural language processing. We will use as an example a hardware shop that sells four items: **hammers**, **nails**, **screws**, and **paint**.

Let's take a look at the following table:

Transaction	Items
1	Nails
2	Hammers and nails
3	Hammers, nails, paint, and screws
4	Hammers, nails, and paint
5	Screws
6	Paint and screws
7	Screws and nails
8	Paint

In this table, we have grouped transactions with items. We could also have grouped each item with the transactions it was involved in. For example, nails were involved in transactions **1**, **2**, **3**, **4**, and **7**, and hammers were involved in **2**, **3**, **4**, and so on. We can also do this with sets of items, for example, **hammers** and **nails** were both involved in transactions **2**, **3**, and **4**. We can write this as the item set `{hammer,nails}` covers the transaction set `[2,3,4]`. There are 16 item sets including the empty set, which covers all transactions.

The relationship between transaction sets forms a lattice structure connecting items with their respective sets. In order to build associative rules, we need to create frequent item sets that exceed the threshold F_T. For example, a frequent item set where F_T = 3 is {screws}, {hammer,nails}, and {paint}. These are simply the items sets that are associated with three or more transactions. The following is a diagram showing part of the lattice from our example. In a similar way, we found the least general generalization in our hypothesis space mapping. Here, we are interested in the lowest boundary of the largest item set. In this example, it is {nails,hammer}.

We can now create association rules of the form *if A then B*, where *A* and *B* are item sets that frequently appear together in a transaction. If we select an edge on this diagram, say the edge between {nails} with a frequency of 5, and {nails, hammer} with a frequency of 3, then we can say that the **confidence** of the association rule *if nails then hammer* is 3/5. Using a frequency threshold together with the confidence of a rule, an algorithm can find all rules that exceed this threshold. This is called **association rule mining**, and it often includes a post-processing phase where unnecessary rules are filtered out, for example—where a more specific rule does not have a higher confidence than a more general parent.

Summary

We began this chapter by exploring a logical language and creating a hypothesis space mapping for a simple example. We discussed the idea of least general generalizations and how to find a path through this space from the most general to the least general hypothesis. We briefly looked at the concept of *learnability*. Next, we looked at tree models and found that they can be applied to a wide range of tasks and are both descriptive and easy to interpret. Trees by themselves, however, are prone to overfitting and the greedy algorithms employed by most tree models can be prone to over-sensitivity to initial conditions. Finally, we discussed both ordered rule lists and unordered rule set-based models. The two different rule models are distinguished by how they handle rule overlaps. The ordered approach is to find a combination of literals that will separate the samples into more homogeneous groups. The unordered approach searches for a hypotheses one class at a time.

In the next chapter, we will look at quite a different type of model—the linear model. These models employ the mathematics of geometry to describe the problem space and, as we will see, form the basis for support vector machines and neural nets.

5
Linear Models

Linear models are one of the most widely used models and form the foundation of many advanced nonlinear techniques such as support vector machines and neural networks. They can be applied to any predictive task such as classification, regression, or probability estimation.

When responding to small changes in the input data, and provided that our data consists of entirely uncorrelated features, linear models tend to be more stable than tree models. As we mentioned in the last chapter, tree models can over-respond to small variations in training data. This is because splits at the root of a tree have consequences that are not recoverable further down the line, that is, producing different branching and potentially making the rest of the tree significantly different. Linear models on the other hand are relatively stable, being less sensitive to initial conditions. However, as you would expect, this has the opposite effect, changing less sensitive data to nuanced data. This is described by the terms **variance** (for over fitting models) and **bias** (for under fitting models). A linear model is typically low-variance and high-bias.

Linear models are generally best approached from a geometric perspective. We know we can easily plot two dimensions of space in a Cartesian co-ordinate system, and we can use the illusion of perspective to illustrate a third. We have also been taught to think of time as being a fourth dimension, but when we start speaking of n dimensions, a physical analogy breaks down. Intriguingly, we can still use many of the mathematical tools that we intuitively apply to three dimensions of space. While it becomes difficult to visualize these extra dimensions, we can still use the same geometric concepts, such as lines, planes, angles, and distance, to describe them. With geometric models, we describe each instance as having a set of real-value features, each of which is a dimension in our geometric space. Let's begin this chapter with a review of the formalism associated with linear models.

We have already disused the basic numerical linear model solution by the least squared method for two variables. It is straightforward and easy to visualize on a 2D coordinate system. When we try to add parameters, as we add features to our model, we need a formalism to replace, or augment, an intuitive visual representation. In this chapter, we will be looking at the following topics:

- The least squares method
- The normal equation method
- Logistic regression
- Regularization

Let's start with the basic model.

Introducing least squares

In a simple one-feature model, our hypothesis function is as follows:

$$h(x) = w_0 + w_1 x$$

If we graph this, we can see that it is a straight line crossing the y axis at w_0 and having a slope of w_1. The aim of a linear model is to find the parameter values that will create a straight line that most closely matches the data. We call these the functions parameter values. We define an objective function, J_w, which we want to minimize:

$$minJ_w = \frac{1}{2m} \sum_{i=1}^{m} \left(h_w\left(x^{(i)}\right) - y^{(i)} \right)^2$$

Here, m is the number of training samples, $h_w(x^{(i)})$ is the estimated value of the ith training sample, and y^i is its actual value. This is the **cost function** of h, because it measures the cost of the error; the greater the error, the higher the cost. This method of deriving the cost function is sometime referred to as the **sum of the squared error** because it sums up the difference between the predicted value and the actual value. This sum is halved as a convenience, as we will see. There are actually two ways that we can solve this. We can either use an iterative gradient descent algorithm or minimize the cost function in one step using the normal equation. We will look at the gradient descent first.
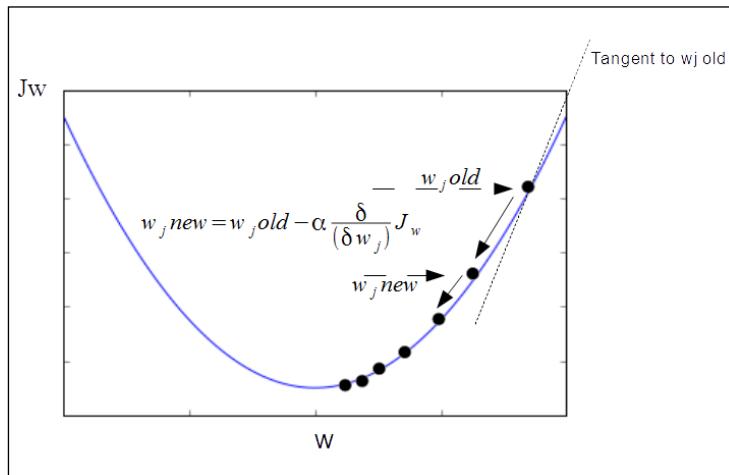
Gradient descent

When we graph parameter values against the cost function, we get a bowl shaped convex function. As parameter values diverge from their optimized values in either direction (from a single minima), the cost of our model grows. As the hypothesis function is linear, the cost function is convex. If this was not the case, then it would be unable to distinguish between **global** and **local minimum**.

The gradient descent algorithm is expressed by the following update rule:

$$\textit{repeat until converges } w_j := w_j - \alpha \frac{\delta}{\left(\delta w_j \right)} J_w$$

Where δ is the first derivative of J_w as it uses the sign of the derivative to determine which way to step. This is simply the sign of the slope of the tangent at each point. The algorithm takes a hyper parameter, a, which is the learning rate that we need to set. It is called a **hyper parameter** to distinguish it from the w parameters that are estimated by our model. If we set the learning rate too small, it will take longer to find the minimum; if set too high, it will overshoot. We may find that we need to run the model several times to determine the best learning rate.

When we apply gradient descent to linear regression, the following formulas, which are the parameters of our model, can be derived. We can rewrite the derivative term to make it easier to calculate. The derivations themselves are quite complex, and it is unnecessary to work through them here. If you know calculus, you will be able to see that the following rules are equivalent. Here, we repeatedly apply two update rules to the hypothesis, employing a stopping function. This is usually when the differences between the parameters on subsequent iterations drop below a threshold, that is, t.

Initialize w_0 and w_1 and repeat:

$$\|wold - wnew\| < t\{$$

$$w_0 : w_0 - a\frac{1}{m}\sum_{i=1}^{m}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)$$

$$w_1 : w_0 - a\frac{1}{m}\sum_{i=1}^{m}\left(\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)x_i\right)$$

$$\}$$

It is important that these update rules are applied simultaneously, that is, they are both applied in the same iteration, so the new values of both w_0 and w_1 are plugged back in the next iteration. This is sometimes called **batch gradient descent** because it updates all the training samples in one *batch*.

It is fairly straightforward to apply these update rules on linear regression problems that have multiple features. This is true if we do not worry about the precise derivations.

For multiple features, our hypothesis function will look like this:

$$h_w\left(x\right) = w^T x = w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

Here, $x_0 = 1$, often called our **bias feature**, is added to help us with the following calculations. We see can see that, by using vectors, we can also write this as simply the transpose of the parameter values multiplied by the feature value vector, x. With multiple feature gradient descents, our cost function will apply to a vector of the parameter values, rather than just a single parameter. This is the new cost function.

$$J\left(w\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)^2$$

J(w) is simply *J(w₀, w₁ …,wₙ)*, where *n* is the number of features. *J* is a function of the parameter vector, *w*. Now, our gradient descent update rule is as follows:

$$update \; w_j \; for \; j = (0,\ldots,n) \left\{ w_j : -w_j - \alpha \frac{1}{m} \alpha \sum_{i=1}^{m} \left(x^{(i)} - y^{(i)} \right) x_j^{(i)} \right\}$$

Notice that we now have multiple features. Therefore, we write the *x* value with the subscript *j* to indicate the *j*ᵗʰ feature. We can break this apart and see that it really represents the *j + 1* nested update rules. Each one is identical, apart from their subscripts, to the training rule that we used for single features.

An important point to mention here, and one that we will revisit in later chapters, is that, to make our models work more efficiently, we can define our own features. For a simple situation, where our hypothesis is to estimate the price of a block of land based on two features, width and depth, obviously, we can multiply these two features to get one feature, that is, area. So, depending on a particular insight that you might have about a problem, it can make more sense to use derived features. We can take this idea further and create our own features to enable our model to fit nonlinear data. A technique to do this is **polynomial regression**. This involves adding power terms to our hypothesis function, making it a polynomial. Here is an example:

$$h_w \left(x \right) = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$

A way to apply this, in the case of our land price example, is to simply add the square and the cube of our *area* feature. There are many possible choices for these terms, and in fact, a better choice in our housing example might be in taking the square root of one of the terms to stop the function exploding to infinity. This highlights an important point, that is, when using polynomial regression, we must be very careful about feature scaling. We can see that the terms in the function get increasingly larger as *x* gets larger.

We now have a model to fit nonlinear data, however, at this stage, we are just manually trying different polynomials. Ideally, we need to be able to incorporate feature selection, to some extent, in our models, rather than have a human try to figure out an appropriate function. We also need to be aware that correlated features may make our models unstable, so we need to devise ways of decomposing correlated features into their components. We look at these aspects in *Chapter 7, Features – How Algorithms See the World*.

The following is a simple implementation of batch gradient descent. Try running it with different values of the learning rate alpha, and on data with a greater bias and/or variance, and also after changing the number of iterations to see what effect this has on the performance of our model:

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

def gradientDescent(x, y, alpha, numIterations):
    xTrans = x.transpose()
    m, n = np.shape(x)
    theta = np.ones(n)
    for i in range(0, numIterations):
        hwx = np.dot(x, theta)
        loss = hwx - y
        cost = np.sum(loss ** 2) / (2 * m)
        print("Iteration %d | Cost: %f " % (i, cost))
        gradient = np.dot(xTrans, loss) / m
        theta = theta - alpha * gradient
    return theta

def genData(numPoints, bias, variance):
    x = np.zeros(shape=(numPoints, 2))
    y = np.zeros(shape=numPoints)
    for i in range(0, numPoints):
        x[i][0] = 1
        x[i][1] = i
        y[i] = (i + bias) + random.uniform(0, 1) * variance
    return x, y

def plotData(x,y,theta):
    plt.scatter(x[...,1],y)
    plt.plot(x[...,1],[theta[0] + theta[1]*xi for xi in x[...,1]])

x, y = genData(20, 25, 10)
```
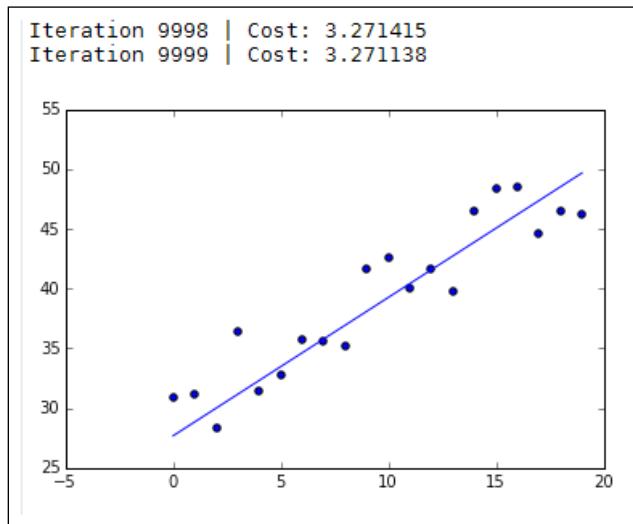
```
iterations= 10000
alpha = 0.001
theta=gradientDescent(x,y,alpha,iterations)
plotData(x,y,theta)
```

The output of the code is as shown in the following screenshot:

This is called **batch gradient descent** because, on each iteration, it updates the parameter values based on all the training samples at once. With **Stochastic gradient descent**, on the other hand, the gradient is approximated by the gradient of a single example at a time. Several passes may be made over the data until the algorithm converges. On each pass, the data is shuffled to prevent it from getting stuck in a loop. Stochastic gradient descent has been successfully applied to large scale learning problems such as natural language processing. One of the disadvantages is that it requires a number of hyper parameters, although this does present opportunities for tweaking such as choosing a loss function or the type of regularization applied. Stochastic gradient descent is also sensitive to feature scaling. Many implementations of this, such as **SGDClassifier** and **SGDRegressor** from the `Sklearn` package, will use an adaptive learning rate by default. This reduces the learning rate as the algorithm moves closer to the minimum. To make these algorithms work well, it is usually necessary to scale the data so that each value in the input vector, *X*, is scaled between 0 and 1 or between -1 and 1. Alternatively, ensure that the data values have a mean of 0 and a variance of 1. This is most easily done using the `StandardScaler` class from `sklearn.preprocessing`.

Gradient descent is not the only algorithm, and in many ways, it is not the most efficient way to minimize the cost function. There are a number of advanced libraries that will compute values for the parameters much more efficiently than if we implemented the gradient descent update rules manually. Fortunately, we do not have to worry too much about the details because there are a number of sophisticated and efficient algorithms for regression already written in Python. For example, in the `sklearn.linear_model` module, there are the **Ridge**, **Lasso**, and **ElasticNet** algorithms that may perform better, depending on your application.

The normal equation

Let's now look at the linear regression problem from a slightly different angle. As I mentioned earlier, there is a numerical solution; thus, rather than iterate through our training set, as we do with gradient descent, we can use what is called the **normal equation** to solve it in one step. If you know some calculus, you will recall that we can minimize a function by taking its derivative and then setting the derivative to zero to solve for a variable. This makes sense because, if we consider our convex cost function, the minimum will be where the slope of the tangent is zero. So, in our simple case with one feature, we differentiate *J(w)* with respect to *w* and set it to zero and solve for *w*. The problem we are interested in is when *w* is an *n +1* parameter vector and the cost function, *J(w)*, is a function of this vector. One way to minimize this is to take the partial derivative of *J(w)* for the parameter values in turn and then set these derivatives to zero, solving for each value of *w*. This gives us the values of *w* that are needed to minimize the cost function.

It turns out that an easy way to solve, what could be a long and complicated calculation, is what is known as the normal equation. To see how this works, we first define a feature matrix, shown as follows:

$$
X = \begin{matrix}
x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\
x_0^{(2)} & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
x_0^{(m)} & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)}
\end{matrix}
$$

This creates an *m* by *n + 1* matrix, where *m* is the number of training examples, and *n* is the number of features. Notice that, in our notation, we now define our training label vector as follows:

$$y = \begin{matrix} y^{(1)} \\ y^{(2)} \\ \ldots \\ y^{(m)} \end{matrix}$$

Now, it turns out that we can compute the parameter values to minimize this cost function by the following equation:

$$w = \left(X^T X \right)^{-1} X^T y$$

This is the normal equation. There are of course many ways to implement this in Python. Here is one simple way using the NumPy `matrix` class. Most implementations will have a regularization parameter that, among other things, prevents an error arising from attempting to transpose a singular matrix. This will occur when we have more features than training data, that is, when *n* is greater than *m*; the normal equation without regularization will not work. This is because the matrix $X^T X$ is non-transposable, and so, there is no way to calculate our term, $(X^T X)^{-1}$. Regularization has other benefits, as we will see shortly:
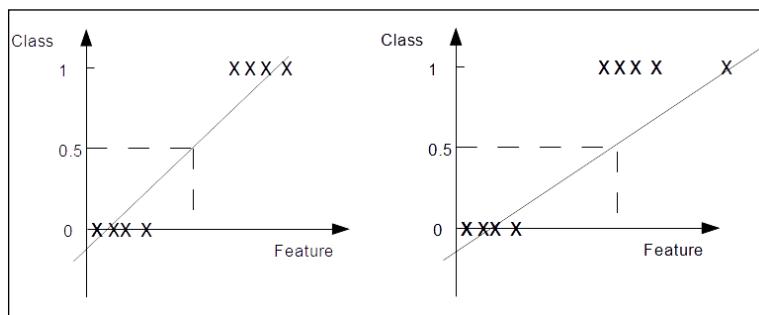
```python
import numpy as np


def normDemo(la=.9):
    X = np.matrix('1 2 5 ; 1 4 6')
    y=np.matrix('8; 16')
    xtrans=X.T
    idx=np.matrix(np.identity(X.shape[1]))
    xti = (xtrans.dot(X)+la * idx).I
    xtidt = xti.dot(xtrans)
    return(xtidt.dot(y))
```

One of the advantages of using the normal equation is that you do not need to worry about feature scaling. Features that have different ranges (for example, if one feature has values between 1 and 10, and another feature has values between zero and 1000) will likely cause problems for gradient descent. Using the normal equation, you do not need to worry about this. Another advantage of the normal equation is that you do not need to choose the learning rate. We saw that, with gradient descent; an incorrectly chosen learning rate could either make the model unnecessarily slow or, if the learning rate is too large, it can cause the model to overshoot the minimum. This may entail an extra step in our testing phase for gradient descent.

The normal equation has its own particular disadvantages; foremost is that it does not scale as well when we have data with a large number of features. We need to calculate the inverse of the transpose of our feature matrix, X. This calculation results in an n by n matrix. Remember that n is the number of features. This actually means that on most platforms the time it takes to invert a matrix grows, approximately, as a cube of n. So, for data with a large number of features, say greater than 10,000, you should probably consider using gradient descent rather than the normal equation. Another problem that arises when using the normal equation is that, when we have more features than training data, that is, when n is greater than m, the normal equation without regularization will not work. This is because the matrix, $X^T X$, is non-transposable, and so there is no way to calculate our term, $(X^T X)\text{-}1$.

Logistic regression

With our least squares model, we have applied it to solve the minimization problem. We can also use a variation of this idea to solve classification problems. Consider what happens when we apply linear regression to a classification problem. Let's take the simple case of binary classification with one feature. We can plot our feature on the x axis against the class labels on the y axis. Our feature variable is continuous, but our target variable on the y axis is discrete. For binary classification, we usually represent a *0* for the negative class, and a *1* for the positive class. We construct a regression line through the data and use a threshold on the y axis to estimate the decision boundary. Here we use a threshold of 0.5.
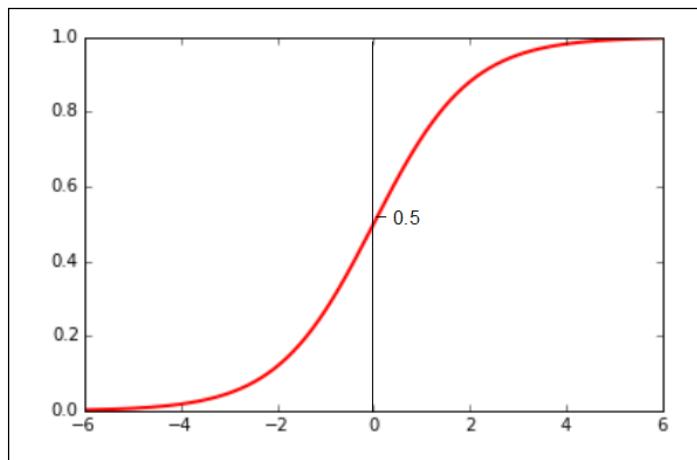
In the figure on the left-hand side, where the variance is small and our positive and negative cases are well separated, we get an acceptable result. The algorithm correctly classifies the training set. In the image on the right-hand side, we have a single outlier in the data. This makes our regression line flatter and shifts our cutoff to the right. The outlier, which clearly belongs in class *1*, should not make any difference to the model's prediction, however, now with the same cutoff point, the prediction misclassifies the first instance of class *1* as class *0*.

One way that we approach the problem is to formulate a different hypothesis representation. For logistic regression, we are going use the linear function as an input to another function, *g*.

$$h_w(x) = g(W^T x) \, where \, 0 \le h_w \le 1$$

The term *g* is called the **sigmoid** or **logistic function**. You will notice from its graph that, on the *y* axis, it has asymptotes at zero and one, and it crosses the axis at *0.5*.

Now, if we replace the *z* with $W^T x$, we can rewrite our hypothesis function like this:

$$h_w(x) = \frac{1}{\left(1 + e^{-w^T x}\right)}$$

As with linear regression, we need to fit the parameters, *w*, to our training data to give us a function that can make predictions. Before we try and fit the model, let's look at how we can interpret the output from our hypothesis function. Since this will return a number between zero and one, the most natural way to interpret this is as it being the probability of the positive class. Since we know, or assume, that each sample can only belong in one of two classes, then the probability of the positive class plus the probability of the negative class must be equal to one. Therefore, if we can estimate the positive class, then we can estimate the probability of the negative class. Since we are ultimately trying to predict the class of a particular sample, we can interpret the output of the hypothesis function as positive if it returns a value greater than or equal to 0.5, or negative otherwise. Now, given the characteristics of the sigmoid function, we can write the following:

$$h_x = g\left(W^T x\right) \geq 0.5 \text{ whenever } W^T x \geq 0$$

Whenever our hypothesis function, on a particular training sample, returns a number greater than or equal to zero, we can predict a positive class. Let's look at a simple example. We have not yet fitted our parameters to this model, and we will do so shortly, but for the sake of this example, let's assume that we have a parameter vector as follows:

$$W = \begin{array}{c} -3 \\ 1 \\ 1 \end{array}$$

Our hypothesis function, therefore, looks like this:

$$h_w\left(x\right) = g\left(-3 + x_1 + x_2\right)$$
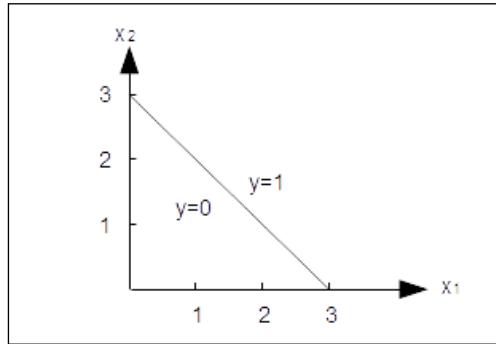
We can predict *y = 1* if the following condition is met:

$$-3 + x_1 + x_2 \geq 0$$

Equivalently:

$$x_1 + x_2 \geq 3$$

This can be sketched with the following graph:

This is simply a straight line between *x=3* and *y=3*, and it represents the **decision boundary**. It creates two regions where we predict either *y = 0* or *y = 1*. What happens when the decision boundary is not a straight line? In the same way that we added polynomials to the hypothesis function in linear regression, we can also do this with logistic regression. Let's write a new hypothesis function with some higher order terms to see how we can fit it to the data:

$$h_w(x) = g\left(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2\right)$$

Here we have added two squared terms to our function. We will see how to fit the parameters shortly, but for now, let's set our parameter vector to the following:

$$w = \begin{array}{c} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{array}$$

So, we can now write the following:

$$Predict\ y = 1\ if\ -1 + x_1^2 + x_2^2 \geq 0$$

Or alternatively, we can write this:

$$Predict \ y = 1 \ if \ x_1^2 + x_2^2 = 1$$

This, you may recognize, is the equation for a circle centered around the origin, and we can use this as our decision boundary. We can create more complex decision boundaries by adding higher order polynomial terms.

The Cost function for logistic regression

Now, we need to look at the important task of fitting the parameters to the data. If we rewrite the cost function we used for linear regression more simply, we can see that the cost is one half of the squared error:
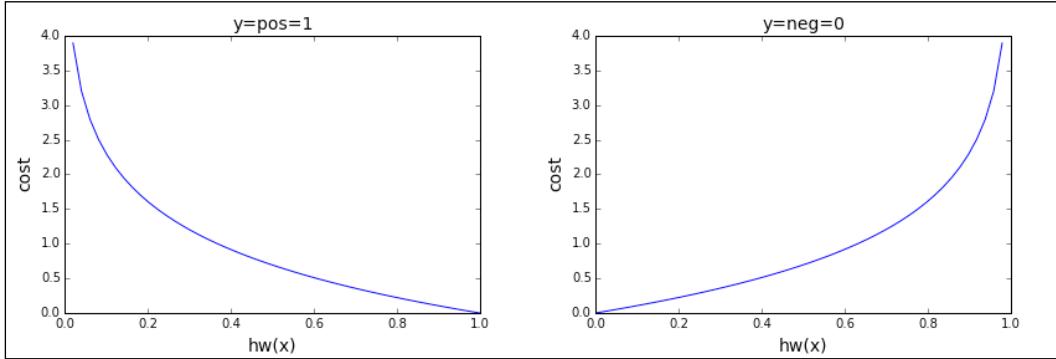
$$Cost\left(h_w\left(x\right), y\right) = \frac{1}{2}\left(h_w\left(x\right) - y\right)^2$$

The interpretation is that it is simply calculating the cost we want the model to incur, given a certain prediction, that is, $h_w(x)$, and a training label, y.

This will work to a certain extent with logistic regression, however, there is a problem. With logistic regression, our hypothesis function is dependent on the nonlinear sigmoid function, and when we plot this against our parameters, it will usually produce a function that is not convex. This means that, when we try to apply an algorithm such as gradient descent to the cost function, it will not necessarily converge to the global minimum. A solution is to define a cost function that is convex, and it turns out that the following two functions, one for each class, are suitable for our purposes:

$$Cost\left(h_w\left(x\right)\right) = -\log\left(h_x\left(w\right)\right) if \ y = 1 Cost\left(h_w\left(x\right)\right) = -\log\left(1 - h_x\left(w\right)\right) if \ y = 0$$

This gives us the following graphs:

Intuitively, we can see that this does what we need it to do. If we consider a single training sample in the positive class, that is *y = 1*, and if our hypothesis function, $h_w(x)$, correctly predicts *1*, then the cost, as you would expect, is *0*. If the output of the hypothesis function is *0*, it is incorrect, so the cost approaches infinity. When *y* is in the negative class, our cost function is the graph on the right. Here the cost is zero when $h_w(x)$ is *0* and rises to infinity when $h_w(x)$ is *1*. We can write this in a more compact way, remembering that *y* is either *0* or *1*:

$$Cost\left(h_w\left(x\right), y\right) = -y\log\left(h_w\left(x\right)\right) - \left(1-y\right)\log\left(1-h_w\left(x\right)\right)$$

We can see that, for each of the possibilities, *y=1* or *y=0*, the irrelevant term is multiplied by *0*, leaving the correct term for each particular case. So, now we can write our cost function as follows:

$$J\left(w\right) = \frac{-1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_w\left(x^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-h_w\left(x^{(i)}\right)\right)\right]$$

So, if we are given a new, unlabeled value of *x*, how do we make a prediction? As with linear regression, our aim is to minimize the cost function, *J(w)*. We can use the same update rule that we used for linear regression, that is, using the partial derivative to find the slope, and when we rewrite the derivative, we get the following:

$$Repeat\ until\ convergance: W_j := W_j - \alpha\sum_{i=1}^{m}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

Multiclass classification

So far, we have just looked at binary classification. For multiclass classification, we assume that each instance belongs to only one class. A slightly different classification problem is where each sample can belong to more than one target class. This is called multi-label classification. We can employ similar strategies on each of these types of problem.

There are two basic approaches:

- One versus all
- One versus many

In the one versus all approach, a single multiclass problem is transformed into a number of binary classification problems. This is called the **one versus all** technique because we take each class in turn and fit a hypothesis function for that particular class, assigning a negative class to the other classes. We end up with different classifiers, each of which is trained to recognize one of the classes. We make a prediction given a new input by running all the classifiers and picking the classifier that predicts a class with the highest probability. To formalize it, we write the following:

$$h_w^{(i)}(x) \text{ for each class } i \text{ predict probability } y = i$$

To make a prediction, we pick the class that maximizes the following:

$$h_w^{(i)}(x)$$

With another approach called the **one versus one** method, a classifier is constructed for each pair of classes. When the model makes a prediction, the class that receives the most votes wins. This method is generally slower than the one versus many method, especially when there are a large number of classes.

All Sklearn classifiers implement multiclass classification. We saw this in *Chapter 2*, *Tools and Techniques*, with the K-nearest neighbors example, where we attempted to predict one of three classes using the iris dataset. Sklearn implements the one versus all algorithm using the `OneVsRestClassifier` class and the one versus one algorithm with `OneVsOneClassifier`. These are called **meta-estimators** because they take another estimator as an input. They have the advantage of being able to permit changing the way more than two classes are handled, and this can result in better performance, either in terms of computational efficiency, or generalization error.

In the following example, we use the SVC:

```
from sklearn import datasets
from sklearn.multiclass import OneVsRestClassifier, OneVsOneClassifier
from sklearn.svm import LinearSVC

X,y = datasets.make_classification(n_samples=10000, n_features=5)
X1,y1 = datasets.make_classification(n_samples=10000, n_features=5)
clsAll=OneVsRestClassifier(LinearSVC(random_state=0)).fit(X, y)
clsOne=OneVsOneClassifier(LinearSVC(random_state=0)).fit(X1, y1)
print("One vs all cost= %f" % clsAll.score(X,y))
print("One vs one cost= %f" % clsOne.score(X1,y1))
```

We will observe the following output:

```
One vs all cost= 0.947400
One vs one cost= 0.949200
```

Regularization

We mentioned earlier that linear regression can become unstable, that is, highly sensitive to small changes in the training data, if features are correlated. Consider the extreme case where two features are perfectly negatively correlated such that any increase in one feature is accompanied by an equivalent decrease in another feature. When we apply our linear regression algorithm to just these two features, it will result in a function that is constant, so this is not really telling us anything about the data. Alternatively, if the features are positively correlated, small changes in them will be amplified. Regularization helps moderate this.

We saw previously that we could get our hypothesis to more closely fit the training data by adding polynomial terms. As we add these terms, the shape of the function becomes more complicated, and this usually results in the hypothesis overfitting the training data and performing poorly on the test data. As we add features, either directly from the data or the ones we derive ourselves, it becomes more likely that the model will overfit the data. One approach is to discard features that we think are less important. However, we cannot know for certain, in advance, what features may contain relevant information. A better approach is to not discard features but rather to shrink them. Since we do not know how much information each feature contains, regularization reduces the magnitude of all the parameters.

We can simply add the term to the cost function.

$$J_w = \frac{1}{2m} \sum_{i=1}^{m} \left(h_w\left(x^{(i)}\right) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^{n} w_j^2$$

The hyper parameter, **lambda,** controls a tradeoff between two goals — the need to fit the training data, and the need to keep the parameters small to avoid overfitting. We do not apply the regularization parameter to our bias feature, so we separate the update rule for the first feature and add a regularization parameter to all subsequent features. We can write it like this:

$$Repeat\ until\ convergance \left\{ w_j := w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_w\left(x^{(i)}\right) - y^{(i)} \right) x_0^{(i)} \right.$$

$$\left. w_j := w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_w\left(x^{(i)}\right) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{m} w_j \right\}$$

Here, we have added our regularization term, $\lambda\, w_j / m$. To see more clearly how this works, we can group all the terms that depend on wj, and our update rule can be rewritten as follows:

$$w_j := w_j \left(1 - \alpha \frac{\lambda}{m} \right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_w\left(x^{(i)}\right) - y^{(i)} \right) x_j^{(i)}$$

The regularization parameter, λ, is usually a small number greater than zero. In order for it to have the desired effect, it is set such that *a λ /m* is a number slightly less than *1*. This will shrink w_j on each iteration of the update.

Now, let's see how we can apply regularization to the normal equation. The equation is as follows:

$$w = \left(X^T X + \lambda I \right)^{-1} X^T y$$

This is sometimes referred to as the **closed form** solution. We add the identity matrix, *I*, multiplied by the regularization parameter. The identity matrix is an *(n+1)* by *(n+1)* matrix consisting of ones on the main diagonal and zeros everywhere else.

In some implementations, we might also make the first entry, the top-left corner, of the matrix zero reflect the fact that we are not applying a regularization parameter to the first bias feature. However, in practice, this will rarely make much difference to our model.

When we multiply it with the identity matrix, we get a matrix where the main diagonal contains the value of λ, with all other positions as zero. This makes sure that, even if we have more features than training samples, we will still be able to invert the matrix X^TX. It also makes our model more stable if we have correlated variables. This form of regression is sometimes called **ridge regression**, and we saw an implementation of this in *Chapter 2, Tools and Techniques*. An interesting alternative to ridge regression is **lasso regression**. It replaces the ridge regression regularization term, $\sum_i wi\ 2$, with $\sum_i \mid wi \mid$. That is, instead of using the sum of the squares of the weights, it uses the sum of the average of the weights. The result is that some of the weights are set to *0* and others are shrunk. Lasso regressions tends to be quite sensitive to the regularization parameter. Unlike ridge regression, lasso regression does not have a closed-form solution, so other forms of numerical optimization need to be employed. Ridge regression is sometimes referred to as using the **L2 norm,** and lasso regularization, the **L1 norm**.

Finally, we will look at how to apply regularization to logistic regression. As with linear regression, logistic regression can suffer from the same problems of overfitting if our hypothesis functions contain higher-order terms or many features. We can modify our logistic regression cost function to add the regularization parameter, as shown as follows:

$$J_w = -\left[\frac{1}{m}\sum_{i=1}^{m} y^{(i)}logh_w\left(x^{(i)}\right)+\left(1-y^{(i)}\right)log\left(1-h_w\left(x^{(i)}\right)\right)\right]+\frac{\lambda}{2}m\sum_{j=1}^{n}w_j^2$$

To implement gradient descent for logistic regression, we end up with an equation that, on the surface, looks identical to the one we used for gradient descent for linear regression. However, we must remember that our hypothesis function is the one we used for logistic regression.

$$w_j := w_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_w\left(x^{(i)}\right)-y^{(i)}\right)x_j^{(i)}+\frac{\lambda}{m}w_j$$

Using the hypothesis function, we get the following:

$$h_w(x) = \frac{1}{\left(1 + e^{-W^T}x\right)}$$

Summary

In this chapter, we studied some of the most used techniques in machine learning. We created hypothesis representations for linear and logistic regression. You learned how to create a cost function to measure the performance of the hypothesis on training data, and how to minimize the cost function in order to fit the parameters, using both gradient descent and the normal equation. We showed how you could fit the hypothesis function to nonlinear data by using polynomial terms in the hypothesis function. Finally, we looked at regularization, its uses, and how to apply it to logistic and linear regression.

These are powerful techniques used widely in many different machine learning algorithms. However, as you have probably realized, there is a lot more to the story. The models we have looked at so far usually require considerable human intervention to get them to perform usefully. For example, we have to set the hyper parameters, such as the learning rate or regularization parameter, and, in the case of non linear data, we have to try and find polynomial terms that will force our hypothesis to fit the data. It will be difficult to determine exactly what these terms are, especially when we have many features. In the next chapter, we will look at the ideas that drive some of the most powerful learning algorithms on the planet, that is, neural networks.

6
Neural Networks

Artificial neural networks, as the name suggests, are based algorithms that attempt to mimic the way neurons work in the brain. Conceptual work began in the 1940s, but it is only somewhat recently that a number of important insights, together with the availability of hardware to run these more computationally expensive models, have given neural networks practical application. They are now state-of-the-art techniques that are at the heart of many advanced machine learning applications.

In this chapter, we will introduce the following topics:

- Logistic units
- The cost function for neural networks
- Implementing a neural network
- Other neural network architectures

Getting started with neural networks

We saw in the last chapter how we could create a nonlinear decision boundary by adding polynomial terms to our hypothesis function. We can also use this technique in linear regression to fit nonlinear data. However, this is not the ideal solution for a number of reasons. Firstly, we have to choose polynomial terms, and for complicated decision boundaries, this can be an imprecise and time-intensive process, which can take quite a bit of trial and error. We also need to consider what happens when we have a large number of features. It becomes difficult to understand exactly how added polynomial terms will change the decision boundary. It also means that the possible number of derived features will grow exponentially. To fit complicated boundaries, we will need many higher-order terms, and our model will become unwieldy, computationally expensive, and hard to understand.

Consider applications such as computer vision, where in a gray scale image, each pixel is a feature that has a value between 0 and 255. For a small image, say 100 pixels by 100 pixels, we have 10,000 features. If we include just quadratic terms, we end up with around 50 million possible features, and to fit complex decision boundaries, we likely need cubic and higher order terms. Clearly, such a model is entirely unworkable.
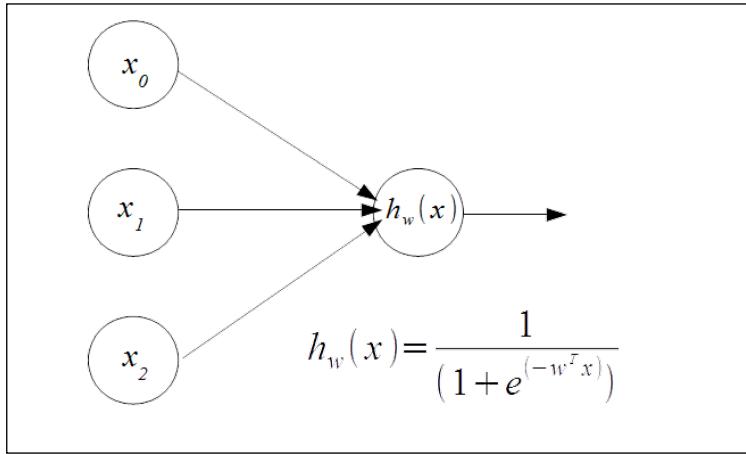
When we approach the problem of trying to mimic the brain, we are faced with a number of difficulties. Considering all the different things that the brain does, we might first think that the brain consists of a number of different algorithms, each specialized to do a particular task, and each hard wired into different parts of the brain. This approach basically considers the brain as a number of subsystems, each with its own program and task. For example, the auditory cortex for perceiving sound has its own algorithm that, for example, does a **Fourier** transform on the incoming sound wave to detect pitch. The visual cortex, on the other hand, has its own distinct algorithm for decoding and converting the signals from the optic nerve into the sense of sight. There is, however, growing evidence that the brain does not function like this at all.

Recent experiments on animals have shown the remarkable adaptabilities of brain tissue. Rewiring the optic nerve to the auditory cortex in animals, scientists found that the brain could learn to see using the machinery of the auditory cortex. The animals were tested to have full vision despite the fact that their visual cortex had been bypassed. It appears that brain tissue, in different parts of the brain, can relearn how to interpret its inputs. So, rather than the brain consisting of specialized subsystems programmed to perform specific tasks, it uses the same algorithm to learn different tasks. This single algorithm approach has many advantages, not least of which is that it is relatively easy to implement. It also means that we can create generalized models and then train them to perform specialized tasks. Like in real brains using a single algorithm to describe how each neuron communicates with the other neurons around it, it allows artificial neural networks to be adaptable and able to carry out multiple higher-level tasks. But, what is the nature of this single algorithm?

When trying to mimic real brain functions, we are forced to greatly simplify many things. For example, there is no way to take into account the role of the chemical state of the brain, or the state of the brain at different stages of development and growth. Most of the neural net models currently in use employ discrete layers of artificial neurons, or units, connected in a well ordered linear sequence or in layers. The brain, on the other hand, consists of many complex, nested, and interconnected neural circuits. Some progress has been made in attempting to imitate these complex feedback systems, and we will look at these at the end of this chapter. However, there is still much that we do not know about real brain action and how to incorporate this complex behavior into artificial neural networks.

Logistic units

As a starting point, we use the idea of a logistic unit over the simplified model of a neuron. It consists of a set of inputs and outputs and an activation function. This activation function is essentially performing a calculation on the set of inputs, and subsequently giving an output. Here, we set the activation function to the sigmoid that we used for logistic regression in the previous chapter:

We have Two input units, x_1 and x_2 and a bias unit, x_0, that is set to one. These are fed into a hypothesis function that uses the sigmoid logistic function and a weight vector, *w*, which parameterizes the hypothesis function. The feature vector, consisting of binary values, and the parameter vector for the preceding example consist of the following:

$$
\begin{array}{cc}
x_0 = 1 & W_0 \\
x = x_1 & W = W_1 \\
x_2 & W_2 \\
x_3 & W_3
\end{array}
$$

To see how we can get this to perform logical functions, let's give the model some weights. We can write this as a function of the sigmoid, *g*, and our weights. To get started, we are just going to choose some weights. We will learn shortly how to train the model to learn its own weights. Let's say that we set out weight such that we have the following hypothesis function:

$$
h_w(x) = g\left(-15 + 10_{x_1} + 10_{x_2}\right)
$$

We feed our model some simple labeled data and construct a truth table:

$$
\begin{array}{cccc}
x_1 & x_2 & y & h_w(x) \\
0 & 0 & 1 & g(-15) \approx 0 \\
0 & 1 & 0 & g(-5) \approx 0 \\
1 & 0 & 0 & g(-5) \approx 0 \\
1 & 1 & 1 & g(5) \approx 1
\end{array}
$$

Although this data appears relatively simple, the decision boundary that is needed to separate the classes is not. Our target variable, y, forms the logical **XNOR** with the input variables. The output is *1* only when both x_1 and x_2 are either *0* or *1*.
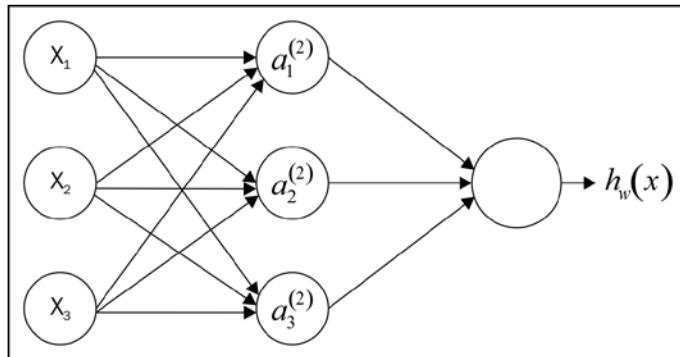
Here, our hypothesis has given us a logical **AND**. That is, it returns a *1* when both x_1 and x_2 are *1*. By setting the weights to other values, we can get our single artificial neuron to form other logical functions.

This gives us the logical **OR** function:

$$ h_w = -5 + 10x_1 + 10x_2 $$

To perform an XNOR, we combine the AND, OR, and NOT functions. To perform negation, that is, a logical **NOT**, we simply choose large negative weights for the input variable that we want to negate.

Logistics units are connected together to form artificial neural networks. These networks consist of an input layer, one or more hidden layers, and an output layer. Each unit has an activation function, here the sigmoid, and is parameterized by the weight matrix W:

We can write out the activation functions for each of the units in the hidden layer:

$$a_1^{(2)} = g\left(W_{10}^{(1)}x_0 + W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3\right)$$

$$a_2^{(2)} = g\left(W_{20}^{(1)}x_0 + W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3\right)$$

$$a_3^{(2)} = g\left(W_{30}^{(1)}x_0 + W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3\right)$$

The activation function for the output layer is as follows:

$$h_w(x) = a_1^{(3)} = g\left(W_{10}^{(2)}a_0^{(2)} + W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)}\right)$$

More generally, we can say a function mapping from a given layer, *j*, to the layer *j+1* is determined by the parameter matrix, W^j. The super script *j* represents the *jth* layer, and the subscript, *i*, denotes the unit in that layer. We denote the parameter or weight matrix, $W^{(j)}$, which governs the mapping from the layer *j* to the layer *j + 1*. We denote the individual weights in the subscript of their matrix index.

Note that the dimensions of the parameter matrix for each layer will be the number of units in the next layer multiplied by the number of units in the current layer plus *1*; this is for x_0, which is the bias layer. More formally, we can write the dimension of the parameter matrix for a given layer, *j*, as follows:

$$d_{(j+1)} \times d_j + 1$$

The subscript *(j + 1)* refers to the number of units in the next input layer and the forward layer, and the $d_j + 1$ refers to the number of units in the current layer plus *1*.

Let's now look at how we can calculate these activation functions using a vector implementation. We can write these functions more compactly by defining a new term, Z, which consists of the weighted linear combination of the input values for each unit on a given layer. Here is an example:

$$a_{(1)}^2 = g\left(Z_1^{(2)}\right)$$

We are just replacing everything in the inner term of our activation function with a single function, Z. Here, the super script *(2)* represents the layer number, and the subscript *1* indicates the unit in that layer. So, more generally, the matrix that defines the activation function for the layer *j* is as follows:

$$= Z_1^{(j)}$$
$$Z^{(j)} = Z_2^{(j)}$$
$$..$$
$$= Z_n^{(j)}$$
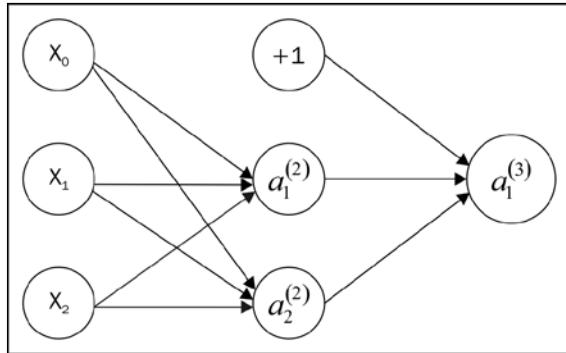
So, in our three layer example, our output layer can be defined as follows:

$$h_w(x) = a^{(3)} = g\big(z(3)\big)$$

We can learn features by first looking at just the three units on the single hidden layer and how it maps its input to the input of the single unit on the output layer. We can see that it is only performing logistic regression using the set of features (a^2). The difference is that now the input features of the hidden layer have themselves been computed using the weights learned from the raw features at the input layer. Through hidden layers, we can start to fit more complicated nonlinear functions.

We can solve our XNOR problem using the following neural net architecture:

Here, we have three units on the input layer, two units plus the bias unit on the single hidden layer, and one unit on the output layer. We can set the weights for the first unit in the hidden layer (not including the bias unit) to perform the logical function x_1 *AND* x_2. The weights for the second unit perform the functions *(NOT x_1)* *AND (NOT x_2)*. Finally, our output layer performs the OR function.

We can write our activation functions as follows:

$$a_1^{(2)} = g\left(-15x_0 + 10x_1 + 10x_2\right)$$
$$a_2^{(2)} = g\left(10x_0 + 20x_1 - 20x_2\right)$$
$$a_1^{(3)} = g\left(-5x_0 + 10x_1 + 10x_2\right)$$

The truth table for this network looks like this:

x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_w(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

To perform multiclass classification with neural networks, we use architectures with an output unit for each class that we are trying to classify. The network outputs a vector of binary numbers with *1* indicating that the class is present. This output variable is an *i* dimensional vector, where *i* is the number of output classes. The output space for four features, for example, would look like this:

$$
y^{(1)} \quad y^{(2)} \quad y^{(3)} \quad y^{(4)}
$$

$y^{(1)}$		$y^{(2)}$		$y^{(3)}$		$y^{(4)}$
1		0		0		0
0	;	1	;	0	;	0
0		0		1		0
0		0		0		1

Our goal is to define a hypothesis function to approximately equal one of these four vectors:

$$h_w(x) \approx y^{(i)}$$

This is essentially a one versus all representation.

We can describe a neural network architecture by the number of layers, *L*, and by the number of units in each layer by a number, s_i, where the subscript indicates the layer number. For convenience, I am going to define a variable, *t*, indicating the number of units on the layer *l + 1*, where *l + 1* is the forward layer, that is, the layer to the right-hand side of the diagram.

Cost function

To fit the weights in a neural net for a given training set, we first need to define a cost function:

$$J_w = \frac{-1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{k} y_k^{(i)} log\left(h_w\left(x^{(i)}\right)\right)_k + \left(1 - y_k^{(i)}\right) log\left(1 - \left(h_w\left(x^{(i)}\right)\right)_k\right) + \frac{\lambda}{2}m\sum_{l=1}^{L-1}\sum_{i=1}^{s}\sum_{j=1}^{t}\left(W_{ji}^{(l)}\right)^2$$

This is very similar to the cost function we used for logistic regression, except that now we are also summing over *k* output units. The triple summation used in the regularization term looks a bit complicated, but all it is really doing is summing over each of the terms in the parameter matrix, and using this to calculate the regularization. Note that the summation, *i*, *l*, and *j* start at *1*, rather than *0*; this is to reflect the fact that we do not apply regularization to the bias unit.

Minimizing the cost function

Now that we have cost function, we need to work out a way to minimize it. As with gradient descent, we need to compute the partial derivatives to calculate the slope of the cost function. This is done using the **back propagation** algorithm. It is called back propagation because we begin by calculating the error at the output layer, then calculating the error for each previous layer in turn. We can use these derivatives calculated by the cost function to work out parameter values for each of the units in our neural network. To do this, we need to define an error term:

$$\delta_j^{(l)} = error\ of\ node\ j\ in\ layer\ |$$

For this example, let's assume that we have a total of three layers, including the input and output layers. The error at the output layer can be written as follows:

$$\delta_j^{(3)} = a_j^{(3)} - y_j = h_w(x) - y_j$$

The activation function in the final layer is equivalent to our hypothesis function, and we can use simple vector subtraction to calculate the difference between the values predicted by our hypothesis, and the actual values in our training set. Once we know the error in our output layer, we are able to *back propagate* to find the error, which is the delta values, in previous layers:

$$\delta^{(2)} = \left(W^{(2)}\right)^T \delta^{(3)} .* g'\left(z^{(2)}\right)$$

This will calculate the error for layer three. We use the transpose of the parameter vector of the current layer, in this example layer 2, multiplied by the error vector from the forward layer, in this case layer 3. We then use pairwise multiplication, indicated by the * symbol, with the derivative of the activation function, *g*, evaluated at the input values given by $z^{(3)}$. We can calculate this derivative term by the following:

$$g'\left(z^{(3)}\right) = a^{(3)} .* \left(1 - a^{(3)}\right)$$

If you know calculus, it is a fairly straight forward procedure to prove this, but for our purposes, we will not go into it here. As you would expect when we have more than one hidden layer, we can calculate the delta values for each hidden layer in exactly the same way, using the parameter vector, the delta vector for the forward layer, and the derivative of the activation function for the current layer. We do not need to calculate the delta values for layer 1 because these are just the features themselves without any errors. Finally, through a rather complicated mathematical proof that we will not go into here, we can write the derivative of the cost function, ignoring regularization, as follows:

$$\frac{\partial}{\left[\partial W_{ij}^{(l)}\right]} J(W) = a_j^{(l)} \delta_i^{(l+1)}$$

By computing the delta terms using back propagation, we can find these partial derivatives for each of the parameter values. Now, let's see how we apply this to a dataset of training samples. We need to define capital delta, Δ, which is just the matrix of the delta terms and has the dimensions, *l:i:j*. This will act as an **accumulator** of the delta values from each node in the neural network, as the algorithm loops through each training sample. Within each loop, it performs the following functions on each training sample:

1. It sets the activation functions in the first layer to each value of *x*, that is, our input features.

2. It performs forward propagation on each subsequent layer in turn up to the output layer to calculate the activation functions for each layer.

3. It computes the delta values at the output layer and begins the process of back propagation. This is similar to the process we performed in forward propagation, except that it occurs in reverse. So, for our output layer in our 3-layer example, it is demonstrated as follows:

$$\delta^{(3)} = a^{(3)} - y^{(i)}$$

Remember that this is all happening in a loop, so we are dealing with one training sample at a time; $y^{(i)}$ represents the target value of the ith training sample. We can now use the back propagation algorithm to calculate the delta values for previous layers. We can now add these values to the accumulator, using the update rule:

$$\Delta^{(l)}_{(ij)} := \Delta^{(l)}_{(ij)} + a^{(l)}_j \delta^{(l+1)}$$

This formula can be expressed in its vectorized form, updating all training samples at once, as shown:

$$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} \left(a^{(l)} \right)^T$$

Now, we can add our regularization term:

$$\Delta^{(i)} := +\Delta^{(i)} + \lambda^{(i)}$$

Finally, we can update the weights by performing gradient descent:

$$W^{(l)} := W^{(l)} - \alpha\Delta^{(l)}$$

Remember that *a* is the learning rate, that is, a hyper parameter we set to a small number between 0 and 1.

Implementing a neural network

There is one more thing we need to consider, and that is the initialization of our weights. If we initialize them to 0, or all to the same number, all the units on the forward layer will be computing the same function at the input, making the calculation highly redundant and unable to fit complex data. In essence, what we need to do is break the symmetry so that we give each unit a slightly different starting point that actually allows the network to create more interesting functions.

Now, let's look at how we might implement this in code. This implementation is written by Sebastian Raschka, taken from his excellent book, *Python Machine Learning*, released by Packt Publishing:

```python
import numpy as np
from scipy.special import expit
import sys


class NeuralNetMLP(object):

    def __init__(self, n_output, n_features, n_hidden=30,
                 l1=0.0, l2=0.0, epochs=500, eta=0.001,
                 alpha=0.0, decrease_const=0.0, shuffle=True,
                 minibatches=1, random_state=None):

        np.random.seed(random_state)
        self.n_output = n_output
        self.n_features = n_features
        self.n_hidden = n_hidden
        self.w1, self.w2 = self._initialize_weights()
        self.l1 = l1
        self.l2 = l2
        self.epochs = epochs
        self.eta = eta
        self.alpha = alpha
```

```
        self.decrease_const = decrease_const
        self.shuffle = shuffle
        self.minibatches = minibatches


    def _encode_labels(self, y, k):

        onehot = np.zeros((k, y.shape[0]))
        for idx, val in enumerate(y):
            onehot[val, idx] = 1.0
        return onehot


    def _initialize_weights(self):
        """Initialize weights with small random numbers."""
        w1 = np.random.uniform(-1.0, 1.0, size=self.n_hidden*(self.n_
features + 1))
        w1 = w1.reshape(self.n_hidden, self.n_features + 1)
        w2 = np.random.uniform(-1.0, 1.0, size=self.n_output*(self.n_
hidden + 1))
        w2 = w2.reshape(self.n_output, self.n_hidden + 1)
        return w1, w2


    def _sigmoid(self, z):

        # return 1.0 / (1.0 + np.exp(-z))
        return expit(z)


    def _sigmoid_gradient(self, z):
        sg = self._sigmoid(z)
        return sg * (1 - sg)


    def _add_bias_unit(self, X, how='column'):

        if how == 'column':
            X_new = np.ones((X.shape[0], X.shape[1]+1))
            X_new[:, 1:] = X
```

```
        elif how == 'row':
            X_new = np.ones((X.shape[0]+1, X.shape[1]))
            X_new[1:, :] = X
        else:
            raise AttributeError('`how` must be `column` or `row`')
        return X_new

    def _feedforward(self, X, w1, w2):

        a1 = self._add_bias_unit(X, how='column')
        z2 = w1.dot(a1.T)
        a2 = self._sigmoid(z2)
        a2 = self._add_bias_unit(a2, how='row')
        z3 = w2.dot(a2)
        a3 = self._sigmoid(z3)
        return a1, z2, a2, z3, a3

    def _L2_reg(self, lambda_, w1, w2):
        """Compute L2-regularization cost"""
        return (lambda_/2.0) * (np.sum(w1[:, 1:] ** 2) + np.sum(w2[:, 1:]
** 2))

    def _L1_reg(self, lambda_, w1, w2):
        """Compute L1-regularization cost"""
        return (lambda_/2.0) * (np.abs(w1[:, 1:]).sum() + np.abs(w2[:,
1:]).sum())

    def _get_cost(self, y_enc, output, w1, w2):

        term1 = -y_enc * (np.log(output))
        term2 = (1 - y_enc) * np.log(1 - output)
        cost = np.sum(term1 - term2)
        L1_term = self._L1_reg(self.l1, w1, w2)
        L2_term = self._L2_reg(self.l2, w1, w2)
        cost = cost + L1_term + L2_term
```

```
            return cost


    def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):

        # backpropagation
        sigma3 = a3 - y_enc
        z2 = self._add_bias_unit(z2, how='row')
        sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)
        sigma2 = sigma2[1:, :]
        grad1 = sigma2.dot(a1)
        grad2 = sigma3.dot(a2.T)

        # regularize
        grad1[:, 1:] += (w1[:, 1:] * (self.l1 + self.l2))
        grad2[:, 1:] += (w2[:, 1:] * (self.l1 + self.l2))

        return grad1, grad2


    def predict(self, X):

        if len(X.shape) != 2:
            raise AttributeError('X must be a [n_samples, n_features]
array.\n'
                                 'Use X[:,None] for 1-feature
classification,'
                                 '\nor X[[i]] for 1-sample
classification')

        a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)
        y_pred = np.argmax(z3, axis=0)
        return y_pred


    def fit(self, X, y, print_progress=False):

        self.cost_ = []
        X_data, y_data = X.copy(), y.copy()
```

```
        y_enc = self._encode_labels(y, self.n_output)

        delta_w1_prev = np.zeros(self.w1.shape)
        delta_w2_prev = np.zeros(self.w2.shape)

        for i in range(self.epochs):

            # adaptive learning rate
            self.eta /= (1 + self.decrease_const*i)

            if print_progress:
                sys.stderr.write('\rEpoch: %d/%d' % (i+1, self.epochs))
                sys.stderr.flush()

            if self.shuffle:
                idx = np.random.permutation(y_data.shape[0])
                X_data, y_data = X_data[idx], y_data[idx]

            mini = np.array_split(range(y_data.shape[0]), self.
minibatches)
            for idx in mini:

                # feedforward
                a1, z2, a2, z3, a3 = self._feedforward(X[idx], self.w1,
self.w2)
                cost = self._get_cost(y_enc=y_enc[:, idx],
                                      output=a3,
                                      w1=self.w1,
                                      w2=self.w2)
                self.cost_.append(cost)

                # compute gradient via backpropagation
                grad1, grad2 = self._get_gradient(a1=a1, a2=a2,
                                                  a3=a3, z2=z2,
                                                  y_enc=y_enc[:, idx],
```
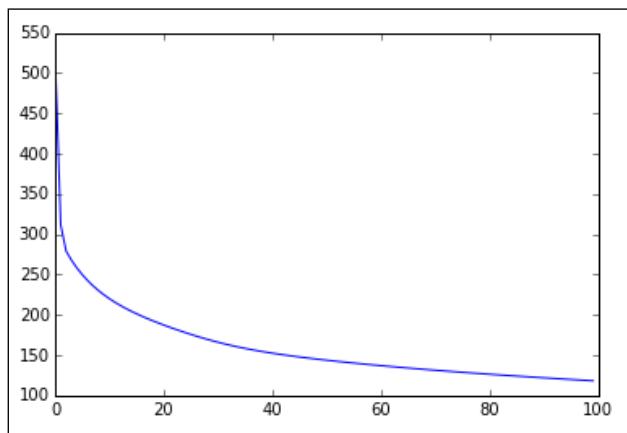
```
                                    w1=self.w1,
                                    w2=self.w2)


            delta_w1, delta_w2 = self.eta * grad1, self.eta * grad2
            self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
            self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
            delta_w1_prev, delta_w2_prev = delta_w1, delta_w2


        return self
```

Now, let's apply this neural net to the iris sample dataset. Remember that this dataset contains three classes, so we set the n_output parameter (the number of output layers) to 3. The shape of the first axis in the dataset refers to the number of features. We create 50 hidden layers and 100 epochs, with each epoch being a complete loop over all the training set. Here, we set the learning rate, alpha, to .001, and we display a plot of the cost against the number of epochs:

```
iris = datasets.load_iris()
X=iris.data
y=iris.target
nn= NeuralNetMLP(3, X.shape[1],n_hidden=50, epochs=100, alpha=.001)
nn.fit(X,y)
plt.plot(range(len(nn.cost_)),nn.cost_)
plt.show()
```
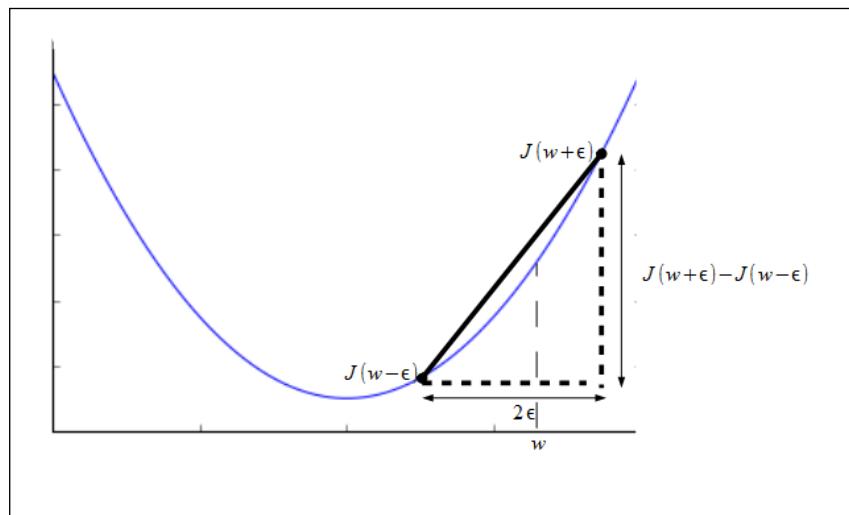
Here is the output:

The graph shows how the cost is decreasing on each epoch. To get a feel for how the model works, spend some time experimenting with it on other data sets and with a variety of input parameters. One particular data set that is used often when testing multiclass classification problems is the MNIST dataset, which is available at `http://yann.lecun.com/exdb/mnist/`. This consists of datasets with 60,000 images of hand drawn letters, along with their labels. It is often used as a benchmark for machine learning algorithms.

Gradient checking

Back propagation, and neural nets in general, are a little difficult to conceptualize. So, it is often not easy to understand how changing any of the model (hyper) parameters will affect the outcome. Furthermore, with different implementations, it is possible to get results that indicate that an algorithm is working correctly, that is, the cost function is decreasing on each level of gradient descent. However, as with any complicated software, there can be hidden bugs that might only manifest themselves under very specific conditions. A way to help eliminate these is through a procedure called **gradient checking**. This is a numerical way of approximating gradients, and we can understand this intuitively by examining the following diagram:

The derivative of *J(w)*, with respect to *w*, can be approximated as follows:

$$\frac{d}{dw} J(w) \approx \frac{\left(J(w+\epsilon) - J(w-\epsilon)\right)}{(2\epsilon)}$$

The preceding formula approximates the derivative when the parameter is a single value. We need to evaluate these derivatives on a cost function, where the weights are a vector. We do this by performing a partial derivative on each of the weights in turn. Here is an example:

$$w = [w_1, w_2, \ldots, w_n]$$

$$\frac{\partial}{(\partial w_n)} J(w) = J(w_1 + \in, w_2, \ldots, w_n) - J(w_{-\in}, w_2, \ldots, w_n)$$

$$\frac{\partial}{(\partial w_n)} J(w) = J(w_1, w_2, \ldots, w_n + \in) - J(w, w_2, \ldots, w_n - \in)$$

Other neural net architectures

Much of the most important work being done in the field of neural net models, and indeed machine learning in general, is using very complex neural nets with many layers and features. This approach is often called **deep architecture** or deep learning. Human and animal learning occurs at a rate and depth that no machine can match. Many of the elements of biological learning still remain a mystery. One of the key areas of research, and one of the most useful in practical applications, is that of object recognition. This is something quite fundamental to living systems, and higher animals have evolved an extraordinary ability to learn complex relationships between objects. Biological brains have many layers; each synaptic event exists in a long chain of synaptic processes. In order to recognize complex objects, such as people's faces or handwritten digits, a fundamental task that is needed is to create a hierarchy of representation from the raw input to higher and higher levels of abstraction. The goal is to transform raw data, such as a set of pixel values, into something we can describe as, say, *a person riding bicycle*. An approach to solving these sorts of problems is to use a sparse representation that creates higher dimensional feature spaces, where there are many features, but only very few of them have non-zero values. This approach is attractive for several reasons. Firstly, features may become more linearly separable in higher feature spaces. Also, it has been shown in certain models that sparsity can be used to make training more efficient and help extract information from very noisy data. We will explore this idea and the general concept of feature extraction in greater detail in the next chapter.

Another interesting idea is that of **recurrent neural networks** or **RNNs**. These are in many ways quite distinct from the feed forward networks that we have considered so far. Rather than simply static mappings between input and output, RNNs have at least one cyclic feedback path. RNNs introduce a time component to the network because a unit's input may include inputs that it received earlier via a feedback loop. All biological neural networks are highly recurrent. Artificial RNNs have shown promise in areas such as speech and hand writing recognition. However, they are, in general, much harder to train because we cannot simply back propagate the error. We have to take into consideration the time component and the dynamic, nonlinear characteristics of such systems. RNNs will provide a very interesting area for future research.

Summary

In this chapter, we introduced the powerful machine learning algorithms of artificial neural networks. We saw how these networks are a simplified model of neurons in the brain. They can perform complex learning tasks, such as learning highly nonlinear decision boundaries, using layers of artificial neurons, or units, to learn new features from labelled data. In the next chapter, we will look at the crucial component of any machine learning algorithm, that is, its features.

7
Features – How Algorithms See the World

So far in this book, we suggested a number of ways and a number of reasons for creating, extracting, or, otherwise, manipulating features. In this chapter, we will address this topic head on. The right features, sometimes called **attributes**, are the central component for machine learning models. A sophisticated model with the wrong features is worthless. Features are how our applications see the world. For all but the most simple tasks, we will process our features before feeding them to a model. There are many interesting ways in which we can do this, and it is such an important topic that it's appropriate to devote an entire chapter to it.

It has only been in the last decade or so that machine learning models have been routinely using tens of thousands of features or more. This allows us to tackle many different problems, such as those where our feature set is large compared to the number of samples. Two typical applications are genetic analysis and text categorization. For genetic analysis, our variables are a set of **gene expression coefficients**. These are based on the number of **mRNA** present in a sample, for example, taken from a tissue biopsy. A classification task might be performed to predict whether a patient has cancer or not. The number of training and test samples together may be a number less than 100. On the other hand, the number of variables in the raw data may range from 6,000 to 60,000. Not only will this translate to a large number of features, it also means that the range of values between features is quite large too. In this chapter, we will cover the following topics:

- Feature types
- Operations and statistics
- Structured features
- Transforming features
- Principle component analysis

Feature types

There are three distinct types of features: quantitative, ordinal, and categorical. We can also consider a fourth type of feature—the Boolean—as this type does have a few distinct qualities, although it is actually a type of categorical feature. These feature types can be ordered in terms of how much information they convey. Quantitative features have the highest information capacity followed by ordinal, categorical, and Boolean.

Let's take a look at the tabular analysis:

Feature type	Order	Scale	Tendency	Dispersion	Shape
Quantitative	Yes	Yes	Mean	Range, variance, and standard deviation	Skewness, kurtosis
Ordinal	Yes	No	Median	Quantiles	NA
Categorical	No	No	Mode	NA	NA

The preceding table shows the three types of features, their statistics, and properties. Each feature inherits the statistics from the features from the next row it in the table. For example, the measurement of central tendency for quantitative features includes the median and mode.

Quantitative features

The distinguishing characteristic of quantitative features is that they are continuous, and they usually involve mapping them to real numbers. Often, feature values can be mapped to a subset of real numbers, for example, expressing age in years; however, care must be taken to use the full scale when calculating statistics, such as mean or standard deviation. Because quantitative features have a meaningful numeric scale, they are often used in geometric models. When they are used in tree models, they result in a binary split, for example, using a threshold value where values above the threshold go to one child and values equal to or below the threshold go to the other child. Tree models are insensitive to monotonic transformations of scale, that is, transformations that do not change the ordering of the feature values. For example, it does not matter to a tree model if we measure length in centimeters or inches, or use a logarithmic or linear scale, we simply have to change the threshold values to the same scale. Tree models ignore the scale of quantitative features and treat them as ordinal. This is also true for rule-based models. For probabilistic models, such as the naïve Bayes classifier, quantitative features need to be discretized into a finite number of bins, and therefore, converted to categorical features.

Ordinal features

Ordinal features are features that have a distinct order but do not have a scale. They can be encoded as integer values; however, doing so does not imply any scale. A typical example is that of house numbers. Here, we can discern the position of a house on a street by its number. We assume that house number 1 will come before house number 20 and that houses with the numbers 10 and 11 would be located close to each other. However, the size of the number does not imply any scale; for example, there is no reason to believe that house number 20 will be larger than house number 1. The domain of an ordinal feature is a totally ordered set such as a set of characters or strings. Because ordinal features lack a linear scale, it does not make sense to add or subtract them; therefore, operations such as averaging ordinal features do not usually make sense or yield any information about the features. Similar to quantitative features in tree models, ordinal features result in a binary split. In general, ordinal features are not readily used in most geometric models. For example, linear models assume a Euclidian instance space where feature values are treated as Cartesian coordinates. For distance-based models, we can use ordinal features if we encode them as integers and the distance between them is simply their difference. This is sometimes referred to as the **hamming distance**.

Categorical features

Categorical features, sometimes called **nominal features**, do not have any ordering or scale, and therefore, they do not allow any statistical summary apart from the mode indicating the most frequent occurrence of a value. Categorical features are often best handled by probabilistic models; however, they can also be used in distance-based models using the hamming distance and by setting the distance to 0 for equal values and 1 for unequal values. A subtype of categorical features is the **Boolean feature,** which maps into the Boolean values of true or false.

Operations and statistics

Features can be defined by the allowable operations that can be performed on them. Consider two features: a person's age and their phone number. Although both these features can be described by integers, they actually represent two very different types of information. This is clear when we see which operations we can usefully perform on them. For example, calculating the average age of a group of people will give us a meaningful result; calculating the average phone number will not.

We can call the range of possible calculations that can be performed on a feature as its statistics. These statistics describe three separate aspects of data. These are — its **central tendency**, its **dispersion**, and its **shape**.

To calculate the central tendency of data, we usually use one or more of the following statistics: the mean (or average), the median (or the middle value in an ordered list), and the mode (or the majority of all values). The mode is the only statistic that can be applied to all data types. To calculate the median, we need feature values that can be somehow ordered, that is ordinal or quantitative. To calculate the mean, values must be expressed on some scale, such as the linear scale. In other words they will need to be quantitative features.

The most common way of calculating dispersion is through the statistics of variance or standard deviation. These are both really the same measure but on different scales, with standard deviation being useful because it is expressed on the same scale as the feature itself. Also, remember that the absolute difference between the mean and the median is never larger than the standard deviation. A simpler statistic for measuring dispersion is the range, which is just the difference between the minimum and maximum values. From here, of course, we can estimate the feature's central tendency by calculating the mid-range point. Another way to measure dispersion is using units such as percentiles or deciles to measure the ratio of instances falling below a particular value. For example, the pth percentile is the value that p percent of instances fall below.

Measuring shape statistics is a little more complicated and can be understood using the idea of the **central moment** of a sample. This is defined as follows:

$$m_k = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^k$$

Here, n is the number of samples, μ is the sample mean, and k is an integer. When $k = 1$, the first central moment is 0 because this is simply the average deviation from the mean, which is always 0. The second central moment is the average squared deviation from the mean, which is the variance. We can define **skewness** as follows:

$$\frac{m_3}{\sigma^3}$$

Here σ is the standard deviation. If this formula gives a value that is positive, then there are more instances with values above the mean rather than below. The data, when graphed, is skewed to the right. When the skew is negative, the converse is true.

We can define **kurtosis** as a similar relationship for the fourth central moment:

$$\frac{m_4}{\sigma^4}$$

It can be shown that a normal distribution has a kurtosis of 3. At values above this, the distribution will be more *peaked*. At kurtosis values below 3, the distribution will be *flatter*.

We previously discussed the three types of data, that is, categorical, ordinal, and quantitative.

Machine learning models will treat the different data types in very distinct ways. For example, a decision tree split on a categorical feature will give rise to as many children as there are values. For ordinal and quantitative features, the splits will be binary, with each parent giving rise to two children based on a threshold value. As a consequence, tree models treat quantitative features as ordinal, ignoring the features scale. When we consider probabilistic models such as the **Bayes classifier**, we can see that it actually treats ordinal features as categorical, and the only way in which it can handle quantitative features is to turn them into a finite number of discrete values, therefore converting them to categorical data.

Geometric models, in general, require features that are quantitative. For example, linear models operate in a Euclidean instance space, with the features acting as Cartesian coordinates. Each feature value is considered as a scalar relationship to other feature values. Distance-based models, such as the k-nearest neighbor, can incorporate categorical features by setting the distance to 0 for equal values and 1 for unequal values. Similarly, we can incorporate ordinal features into distance-based models by counting the number of values between two values. If we are encoding feature values as integers, then the distance is simply the numerical difference. By choosing an appropriate distance metric, it is possible to incorporate ordinal and categorical features into distance-based models.

Structured features

We assume that each instance can be represented as a vector of feature values and that all relevant aspects are represented by this vector. This is sometimes called an **abstraction** because we filter out unnecessary information and represent a real-world phenomena with a vector. For example, representing the entire works of Leo Tolstoy as a vector of word frequencies is an abstraction. We make no pretense that this abstraction will serve any more than a very particular limited application. We may learn something about Tolstoy's use of language and perhaps elicit some information regarding the sentiment and subject of Tolstoy's writing. However, we are unlikely to gain any significant insights into the broad canvas of the 19th century Russia portrayed in these works. A human reader, or a more sophisticated algorithm, will gain these insights not from the counting of each word but by the structure that these words are part of.

We can think of structured features in a similar way to how we may think about queries in a database programming language, such as SQL. A SQL query can represent an aggregation over variables to do things such as finding a particular phrase or finding all the passages involving a particular character. What we are doing in a machine learning context is creating another feature with these aggregate properties.

Structured features can be created prior to building the model or as part of the model itself. In the first case, the process can be understood as being a translation from the first order logic to a propositional logic. A problem with this approach is that it can create an explosion in the number of potential features as a result of combinations with existing features. Another important point is that, in the same way that in SQL one clause can cover a subset of another clause, structural features can also be logically related. This is exploited in the branch of machine learning that is particularly well suited to natural language processing, known as **inductive logic programming**.

Transforming features

When we transform features, our aim, obviously, is to make them more useful to our models. This can be done by adding, removing, or changing information represented by the feature. A common feature transformation is that of changing the feature type. A typical example is **binarization**, that is, transforming a categorical feature into a set of binary ones. Another example is changing an ordinal feature into a categorical feature. In both these cases, we lose information. In the first instance, the value of a single categorical feature is mutually exclusive, and this is not conveyed by the binary representation. In the second instance, we lose the ordering information. These types of transformations can be considered inductive because they consist of a well-defined logical procedure that does not involve an objective choice apart from the decision to carry out these transformations in the first place.

Binarization can be easily carried out using the `sklearn.preprocessing.Binarizer` module. Let's take a look at the following commands:

```
from sklearn.preprocessing import Binarizer
from random import randint
bin=Binarizer(5)
X=[randint(0,10) for b in range(1,10)]
print(X)
print(bin.transform(X))
```

The following is the output for the preceding commands:

```
[5, 6, 1, 7, 5, 3, 3, 3, 7]
[[0 1 0 1 0 0 0 0 1]]
```

Features that are categorical often need to be encoded into integers. Consider a very simple dataset with just one categorical feature, City, with three possible values, Sydney, Perth, and Melbourne, and we decide to encode the three values as 0, 1, and 2, respectively. If this information is to be used in a linear classifier, then we write the constraint as a linear inequality with a weight parameter. The problem, however, is that this weight cannot encode for a three way choice. Suppose we have two classes, east coast and west coast, and we need our model to come up with a decision function that will reflect the fact that Perth is on the west coast and both Sydney and Melbourne are on the east coast. With a simple linear model, when the features are encoded in this way, then the decision function cannot come up with a rule that will put Sydney and Melbourne in the same class. The solution is to blow up the feature space to three features, each getting their own weights. This is called **one hot encoding**. Sciki-learn implements the `OneHotEncoder()` function to perform this task. This is an estimator that transforms each categorical feature, with m possible values into m binary features. Consider that we are using a model with data that consists of the city feature as described in the preceding example and two other features—gender, which can be either male or female, and an occupation, which can have three values—doctor, lawyer, or banker. So, for example, *a female banker from Sydney* would be represented as *[1,2,0]*. Three more samples are added for the following example:

```
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder()
enc.fit([[1,2,0], [1, 1, 0], [0, 2, 1], [1, 0, 2]])
print(enc.transform([1,2,0]).toarray())
```

We will get the following output:

$$\begin{bmatrix} [0. & 1. & 0. & 0. & 1. & 1. & 0. & 0.] \end{bmatrix}$$

Since we have two genders, three cities, and three jobs in this dataset, the first two numbers in the transform array represent the gender, the next three represent the city, and the final three represent the occupation.

Discretization

I have already briefly mentioned the idea of thresholding in relation to decision trees, where we transform an ordinal or quantitative feature into a binary feature by finding an appropriate feature value to split on. There are a number of methods, both supervised and unsupervised, that can be used to find an appropriate split in continuous data, for example, using the statistics of central tendency (supervised), such as the mean or median or optimizing an objective function based on criteria such as information gain.

We can go further and create multiple thresholds, transforming a quantitative feature into an ordinal one. Here, we divide a continuous quantitative feature into numerous discrete ordinal values. Each of these values is referred to as a **bin**, and each bin represents an interval on the original quantitative feature. Many machine learning models require discrete values. It becomes easier and more comprehensible to create rule-based models using discrete values. Discretization also makes features more compact and may make our algorithms more efficient.

One of the most common approaches is to choose bins such that each bin has approximately the same number of instances. This is called **equal frequency discretization**, and if we apply it to just two bins, then this is the same as using the median as a threshold. This approach can be quite useful because the bin boundaries can be set up in such a way that they represent quantiles. For example, if we have 100 bins, then each bin represents a percentile.

Alternatively, we can choose the boundaries so that each bin has the same interval width. This is called **equal width discretization**. A way of working out the value of this bin's width interval is simply to divide the feature range by the number of bins. Sometimes, the features do not have an upper or lower limit, and we cannot calculate its range. In this case, integer numbers of standard deviations above and below the mean can be used. Both width and frequency discretization are unsupervised. They do not require any knowledge of the class labels to work.

Let's now turn our attention to supervised discretization. There are essentially two approaches: the top-down or **divisive**, and the **agglomerative** or bottom-up approach. As the names suggest, divisive works by initially assuming that all samples belong to the same bin and then progressively splits the bins. Agglomerative methods begin with a bin for each instance and progressively merges these bins. Both methods require some stopping criteria to decide if further splits are necessary.

The process of recursively partitioning feature values through thresholding is an example of divisive discretization. To make this work, we need a scoring function that finds the best threshold for particular feature values. A common way to do this is to calculate the information gain of the split or its entropy. By determining how many positive and negative samples are covered by a particular split, we can progressively split features based on this criterion.

Simple discretization operations can be carried out by the Pandas **cut** and **qcut** methods. Consider the following example:

```python
import pandas as pd

import numpy as np

print(pd.cut(np.array([1,2,3,4]), 3, retbins = True, right = False))
```

Here is the output observed:

```
([[1, 2), [2, 3), [3, 4.003), [3, 4.003)]
Categories (3, object): [[1, 2) < [2, 3) < [3, 4.003)], array([ 1.   , 2.   ,
3.   , 4.003]))
```

Normalization

Thresholding and discretization, both, remove the scale of a quantitative feature and, depending on the application, this may not be what we want. Alternatively, we may want to add a measure of scale to ordinal or categorical features. In an unsupervised setting, we refer to this as **normalization**. This is often used to deal with quantitative features that have been measured on a different scale. Feature values that approximate a normal distribution can be converted to *z* scores. This is simply a signed number of standard deviations above or below the mean. A positive *z* score indicates a number of standard deviations above the mean, and a negative *z* score indicates the number of standard deviations below the mean. For some features, it may be more convenient to use the variance rather than the standard deviation.

A stricter form of normalization expresses a feature on a 0 to 1 scale. If we know a features range, we can simply use a linear scaling, that is, divide the difference between the original feature value and the lowest value with the difference between the lowest and highest value. This is expressed in the following:

$$f_n = \frac{(f - l)}{(h - l)}$$

Here, f_n is the normalized feature, f is the original feature, and l and h are the lowest and highest values, respectively. In many cases, we may have to guess the range. If we know something about a particular distribution, for example, in a normal distribution more than 99% of values are likely to fall within +3 or -3 standard deviations of the mean, then we can write a linear scaling such as the following:

$$f_n = \frac{(f - \mu)}{(6\sigma)} + \frac{1}{2}$$

Here, μ is the mean and σ is the standard deviation.

Calibration

Sometimes, we need to add scale information to an ordinal or categorical feature. This is called feature **calibration**. It is a supervised feature transformation that has a number of important applications. For example, it allows models that require scaled features, such as linear classifiers, to handle categorical and ordinal data. It also gives models the flexibility to treat features as ordinal, categorical, or quantitative. For binary classification, we can use the posterior probability of the positive class, given a features value, to calculate the scale. For many probabilistic models, such as naive Bayes, this method of calibration has the added advantage in that the model does not require any additional training once the features are calibrated. For categorical features, we can determine these probabilities by simply collecting the relative frequencies from a training set.

There are cases where we might need to turn quantitative or ordinal features in to categorical features yet maintain an ordering. One of the main ways we do this is through a process of **logistic calibration**. If we assume that the feature is normally distributed with the same variance, then it turns out that we can express a likelihood ratio, the ration of positive and negative classes, given a feature value v, as follows:

$$LR(v) = \frac{(P(v \mid pos))}{(P(v \mid neg))} = exp(d'z)$$

Where d prime is the difference between the means of the two classes divided by the standard deviation:

$$d' = \frac{(\mu pos - \mu neg)}{(\sigma)}$$

Also, z is the z score:

$$\frac{(v - \mu)}{\sigma} \text{ assuming equal class distribution} : \mu = \frac{(\mu pos + \mu neg)}{2}$$

To neutralize the effect of nonuniform class distributions, we can calculate calibrated features using the following:

$$F_c(x) = \frac{(LRF(x))}{(1 + LR(F(x)))} = exp \frac{(d'z(x))}{(1 + exp(d'z(x)))}$$

This, you may notice, is exactly the sigmoid activation function we used for logistic regression. To summarize logistic calibration, we essentially use three steps:

1. Estimate the class means for the positive and negative classes.

2. Transform the features into z scores.

3. Apply the sigmoid function to give calibrated probabilities.

Sometimes, we may skip the last step, specifically if we are using distance-based models where we expect the scale to be additive in order to calculate Euclidian distance. You may notice that our final calibrated features are multiplicative in scale.
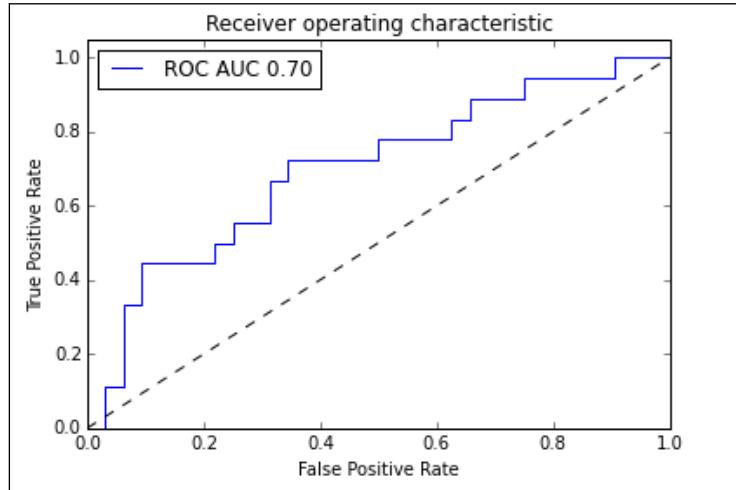
Another calibration technique, **isotonic calibration**, is used on both quantitative and ordinal features. This uses what is known as a **ROC** curve (stands for **Receiver Operator Characteristic**) similar to the coverage maps used in the discussion of logical models in *Chapter 4, Models – Learning from Information*. The difference is that with an ROC curve, we normalize the axis to *[0,1]*.

We can use the `sklearn` package to create an ROC curve:

```python
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier


X, y = datasets.make_classification(n_samples=100,n_classes=3,n_
features=5, n_informative=3, n_redundant=0,random_state=42)
# Binarize the output
y = label_binarize(y, classes=[0, 1, 2])
n_classes = y.shape[1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.5)
classifier = OneVsRestClassifier(svm.SVC(kernel='linear',
probability=True, ))
y_score = classifier.fit(X_train, y_train).decision_function(X_test)
fpr, tpr, _ = roc_curve(y_test[:,0], y_score[:,0])
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot(fpr, tpr, label='ROC AUC %0.2f' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic')
plt.legend(loc="best")
plt.show()
```

Here is the output observed:

The ROC curve maps the true positive rates against the false positive rate for different threshold values. In the preceding diagram, this is represented by the dotted line. Once we have constructed the ROC curve, we calculate the number of positives, m_i, and the total number of instances, n_i, in each segment of the convex hull. The following formula is then used to calculate the calibrated feature values:

$$v_c = \frac{(m_i + 1)}{(m_i + 1 + c(n_i - m_i + 1))}$$

In this formula, c is the prior odds, that is, the ratio of the probability of the positive class over the probability of the negative class.

So far in our discussion on feature transformations, we assumed that we know all the values for every feature. In the real world, this is often not the case. If we are working with probabilistic models, we can estimate the value of a missing feature by taking a weighted average over all features values. An important consideration is that the existence of missing feature values may be correlated with the target variable. For example, data in an individual's medical history is a reflection of the types of testing that are performed, and this in turn is related to an assessment on risk factors for certain diseases.

If we are using a tree model, we can randomly choose a missing value, allowing the model to split on it. This, however, will not work for linear models. In this case, we need to fill in the missing values through a process of **imputation**. For classification, we can simply use the statistics of the mean, median, and mode over the observed features to impute the missing values. If we want to take feature correlation into account, we can construct a predictive model for each incomplete feature to predict missing values.

Since scikit-learn estimators always assume that all values in an array are numeric, missing values, either encoded as blanks, NaN, or other placeholders, will generate errors. Also, since we may not want to discard entire rows or columns, as these may contain valuable information, we need to use an imputation strategy to complete the dataset. In the following code snippet, we will use the `Imputer` class:

```
from sklearn.preprocessing import Binarizer, Imputer, OneHotEncoder
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
print(imp.fit_transform([[1, 3], [4, np.nan], [5, 6]]))
```

Here is the output:

```
[[ 1.    3. ]
 [ 4.    4.5]
 [ 5.    6. ]]
```

Many machine learning algorithms require that features are **standardized**. This means that they will work best when the individual features look more or less like normally distributed data with near-zero mean and unit variance. The easiest way to do this is by subtracting the mean value from each feature and scaling it by dividing by the standard deviation. This can be achieved by the `scale()` function or the `standardScaler()` function in the `sklearn.preprocessing()` function. Although these functions will accept sparse data, they probably should not be used in such situations because centering sparse data would likely destroy its structure. It is recommended to use the `MacAbsScaler()` or `maxabs_scale()` function in these cases. The former scales and translates each feature individually by its maximum absolute value. The latter scales each feature individually to a range of *[-1,1]*. Another specific case is when we have outliers in the data. In these cases using the `robust_scale()` or `RobustScaler()` function is recommended.

Often, we may want to add complexity to a model by adding polynomial terms. This can be done using the `PolynomialFeatures()` function:

```
from sklearn.preprocessing import PolynomialFeatures
X=np.arange(9).reshape(3,3)
```

```
poly=PolynomialFeatures(degree=2)
print(X)
print(poly.fit_transform(X))
```

We will observe the following output:

$$\begin{bmatrix} \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} \\ \begin{bmatrix} 3 & 4 & 5 \end{bmatrix} \\ \begin{bmatrix} 6 & 7 & 8 \end{bmatrix} \end{bmatrix}$$
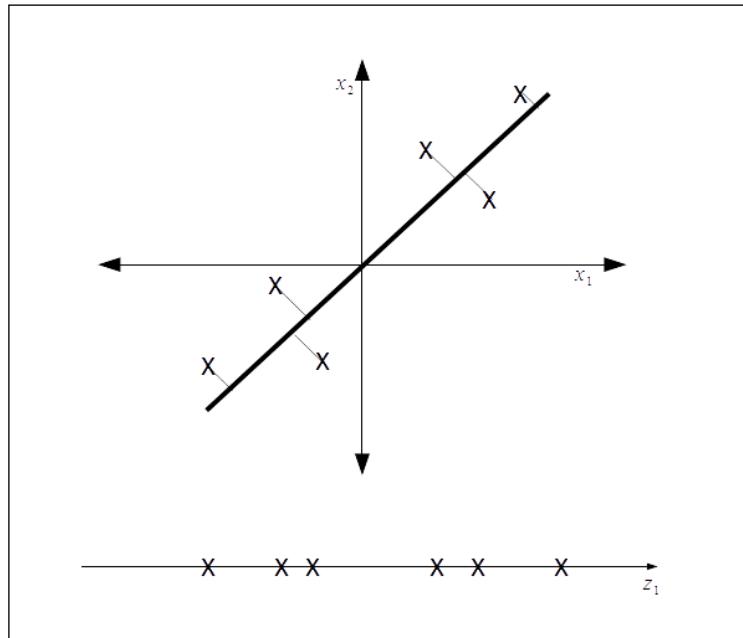
$$\begin{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 2 & 0 & 0 & 0 & 1 & 2 & 4 \end{bmatrix} \\ \begin{bmatrix} 1 & 3 & 4 & 5 & 9 & 12 & 15 & 16 & 20 & 25 \end{bmatrix} \\ \begin{bmatrix} 1 & 6 & 7 & 8 & 36 & 42 & 48 & 49 & 56 & 64 \end{bmatrix} \end{bmatrix}$$

Principle component analysis

Principle Component Analysis (**PCA**) is the most common form of dimensionality reduction that we can apply to features. Consider the example of a dataset consisting of two features and we would like to convert this two-dimensional data into one dimension. A natural approach would be to draw a line of the closest fit and project each data point onto this line, as shown in the following graph:

PCA attempts to find a surface to project the data by minimizing the distance between the data points and the line we are attempting to project this data to. For the more general case where we have *n* dimensions and we want to reduce this space to k-dimensions, we find *k* vectors *u(1),u(2), ..., u(k)* onto which to project the data so as to minimize the projection error. That is we are trying to find a k-dimensional surface to project the data.

This looks superficially like linear regression however it is different in several important ways. With linear regression we are trying to predict the value of some output variable given an input variable. In PCA we are not trying to predict an output variable, rather we are trying to find a subspace onto which to project our input data. The error distances, as represented in the preceding graph, is not the vertical distance between a point and the line, as is the case for linear regression, but rather the closest orthogonal distance between the point and the line. Thus, the error lines are at an angle to the axis and form a right angle with our projection line.

An important point is that in most cases, PCA requires that the features are scaled and are mean normalized, that is, the features have zero mean and have a comparable range of values. We can calculate the mean using the following formula:

$$\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)}$$

The sum is calculated by replacing the following:

$$x_j^{(i)} \text{ with } x_j - \mu_j$$

If the features have scales that are significantly different, we can rescale using the following:

$$\frac{\left(x_j - \mu_j\right)}{\sigma_j}$$

These functions are available in the `sklearn.preprocessing` module.

The mathematical process of calculating both the lower dimensional vectors and the points on these vectors where we project our original data involve first calculating the covariance matrix and then calculating the eigenvectors of this matrix. To calculate these values from first principles is quite a complicated process. Fortunately, the `sklearn` package has a library for doing just this:

```python
from sklearn.decomposition import PCA
import numpy as np
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
pca = PCA(n_components=1)
pca.fit(X)
print(pca.transform(X))
```

We will get the following output:

$$\begin{bmatrix} [-1.3834058] \\ [-2.22189802] \\ [-3.60530382] \\ [1.3834058 \quad] \\ [2.22189802 \quad] \\ [3.60530382 \quad] \end{bmatrix}$$

Summary

There are a rich variety of ways in which we can both transform and construct new features to make our models work more efficiently and give more accurate results. In general, there are no hard and fast rules for deciding which of the methods to use for a particular model. Much depends on the feature types (quantitative, ordinal, or categorical) that you are working with. A good first approach is to normalize and scale the features, and if the model requires it, transform the feature to an appropriate type, as we do through discretization. If the model performs poorly, it may be necessary to apply further preprocessing such as PCA. In the next chapter, we will look at ways in which we can combine different types of models, through the use of ensembles, to improve performance and provide greater predictive power.

8
Learning with Ensembles

The motivation for creating machine learning ensembles comes from clear intuitions and is grounded in a rich theoretical history. Diversity, in many natural and human-made systems, makes them more resilient to perturbations. Similarly, we have seen that averaging results from a number of measurements can often result in a more stable models that are less susceptible to random fluctuations, such as outliers or errors in data collection.

In this chapter, we will divide this rather large and diverse space into the following topics:

- Ensemble types
- Bagging
- Random forests
- Boosting

Ensemble types

Ensemble techniques can be broadly divided into two types:

- **Averaging method**: This is the method in which several estimators are run independently and their predictions are averaged. This includes random forests and bagging methods.

- **Boosting method**: This is the method in which weak learners are built sequentially using weighted distributions of the data based on the error rates.

Ensemble methods use multiple models to obtain better performance than any single constituent model. The aim is to not only build diverse and robust models, but also work within limitations, such as processing speed and return times. When working with large datasets and quick response times, this can be a significant developmental bottleneck. Troubleshooting and diagnostics are an important aspect of working with all machine learning models, but especially when we are dealing with models that may take days to run.

The types of machine learning ensembles that can be created are as diverse as the models themselves, and the main considerations revolve around three things: how we divide our data, how we select the models, and the methods we use to combine their results. This simplistic statement actually encompasses a very large and diverse space.

Bagging

Bagging, also called **bootstrap aggregating**, comes in a few flavors and these are defined by the way they draw random subsets from the training data. Most commonly, bagging refers to drawing samples with replacement. Because the samples are replaced, it is possible for the generated datasets to contain duplicates. It also means that data points may be excluded from a particular generated dataset, even if this generated set is the same size as the original. Each of the generated datasets will be different and this is a way to create diversity among the models in an ensemble. We can calculate the probability that a data point is not selected in a sample using the following example:

$$\left(1 - \frac{1}{n}\right)^n$$

Here, n is the number of bootstrap samples. Each of the n bootstrap samples results in a different hypothesis. The class is predicted either by averaging the models or by choosing the class predicted by the majority of models. Consider an ensemble of linear classifiers. If we use majority voting to determine the predicted class, we create a piece-wise linear classifier boundary. If we transform the votes to probabilities, then we partition the instance space into segments that can each potentially have a different score.

It should also be mentioned that it is possible, and sometimes desirable, to use random subsets of features; this is called **subspace sampling**. Bagging estimators work best with complex models such as fully developed decision trees because they can help reduce overfitting. They provide a simple, out-of-the-box, way to improve a single model.

Scikit-learn implements a `BaggingClassifier` and `BaggingRegressor` objects. Here are some of their most important parameters:

Parameter	Type	Description	Default
`base_estimator`	Estimator	This is the model the ensemble is built on.	Decision tree
`n_estimators`	Int	This is the number of base estimators.	10
`max_samples`	Int or float	This is the number of samples to draw. If float draw `max_samples*X.shape[0]`.	1.0
`max_features`	Int or float	This is the number of features to draw. If float draw `max_features*X.shape[1]`.	1.0
`bootstrap`	Boolean	These are the samples drawn with replacement.	True
`bootstrap_features`	Boolean	These are the features drawn with replacement.	False

As an example, the following snippet instantiates a bagging classifier comprising of 50 decision tree classifier base estimators each built on random subsets of half the features and half the samples:

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets

bcls=BaggingClassifier(DecisionTreeClassifier(),max_samples=0.5, max_features=0.5, n_estimators=50)
X,y=datasets.make_blobs(n_samples=8000,centers=2, random_state=0, cluster_std=4)
bcls.fit(X,y)
print(bcls.score(X,y))
```

Random forests

Tree-based models are particularly well suited to ensembles, primarily because they can be sensitive to changes in the training data. Tree models can be very effective when used with **subspace sampling**, resulting in more diverse models and, since each model in the ensemble is working on only a subset of the features, it reduces the training time. This builds each tree using a different random subset of the features and is therefore called a **random forest**.

A random forest partitions an instance space by finding the intersection of the partitions in the individual trees in the forest. It defines a partition that can be finer, that is, will take in more detail, than a partition created by any individual tree in the forest. In principle, a random forest can be mapped back to an individual tree, since each intersection corresponds to combining the branches of two different trees. The random forest can be thought of as essentially an alternative training algorithm for tree-based models. A linear classifier in a bagging ensemble is able to learn a complicated decision boundary that would be impossible for a single linear classifier to learn.

The `sklearn.ensemble` module has two algorithms based on decision trees, random forests and extremely randomized trees. They both create diverse classifiers by introducing randomness into their construction and both include classes for classification and regression. With the `RandomForestClassifier` and `RandomForestRegressor` class each tree is built using bootstrap samples. The split chosen by the model is not the best split among all features, but is chosen from a random subset of features.

Extra trees

The `extra trees` method, as with random forests, uses a random subset of features, but instead of using the most discriminative thresholds, the best of a randomly generated set of thresholds is used. This acts to reduce variance at the expense of a small increase in bias. The two classes are `ExtraTreesClassifier` and `ExtraTreesRegressor`.

Let's take a look at an example of the `random forest` classifier and the `extra trees` classifier. In this example, we use `VotingClassifier` to combine different classifiers. The voting classifier can help balance out an individual model's weakness. In this example, we pass four weights to the function. These weights determine each individual model's contribution to the overall result. We can see that the two tree models overfit the training data, but also tend to perform better on the test data. We can also see that `ExtraTreesClassifier` achieved slightly better results on the test set compared to the `RandomForest` object. Also, the `VotingClasifier` object performed better on the test set than all its constituent classifiers. It is worth, while running this with different weightings as well as on different datasets, seeing how the performance of each model changes:

```
from sklearn import cross_validation
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
```

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import VotingClassifier
from sklearn import datasets


def vclas(w1,w2,w3, w4):


    X , y = datasets.make_classification(n_features= 10, n_informative=4,
n_samples=500, n_clusters_per_class=5)
    Xtrain,Xtest, ytrain,ytest= cross_validation.train_test_
split(X,y,test_size=0.4)


    clf1 = LogisticRegression(random_state=123)
    clf2 = GaussianNB()
    clf3 = RandomForestClassifier(n_estimators=10,bootstrap=True, random_
state=123)
    clf4= ExtraTreesClassifier(n_estimators=10, bootstrap=True,random_
state=123)


    clfes=[clf1,clf2,clf3,clf4]


    eclf = VotingClassifier(estimators=[('lr', clf1), ('gnb', clf2),
('rf', clf3),('et',clf4)],
                            voting='soft',
                            weights=[w1, w2, w3,w4])



    [c.fit(Xtrain, ytrain) for c in (clf1, clf2, clf3,clf4, eclf)]


    N = 5
    ind = np.arange(N)
    width = 0.3
    fig, ax = plt.subplots()


    for i, clf in enumerate(clfes):
        print(clf,i)
```
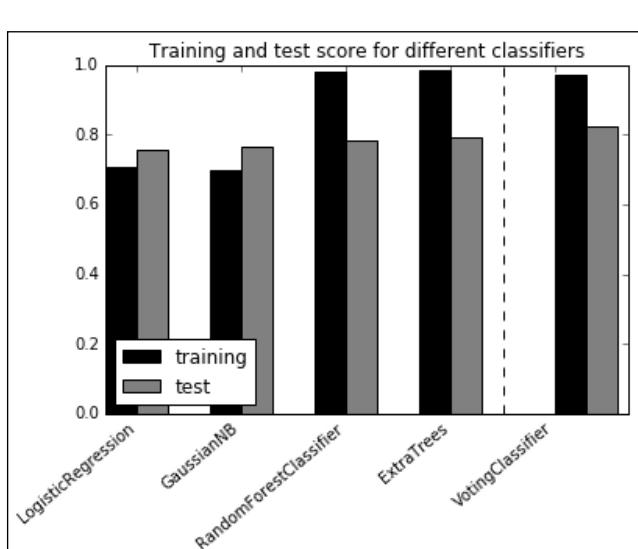
```
        p1=ax.bar(i,clfes[i].score(Xtrain,ytrain,),
width=width,color="black")

        p2=ax.bar(i+width,clfes[i].score(Xtest,ytest,),
width=width,color="grey")

    ax.bar(len(clfes)+width,eclf.score(Xtrain,ytrain,),
width=width,color="black")

    ax.bar(len(clfes)+width *2,eclf.score(Xtest,ytest,),
width=width,color="grey")

    plt.axvline(3.8, color='k', linestyle='dashed')

    ax.set_xticks(ind + width)

    ax.set_xticklabels(['LogisticRegression',

                        'GaussianNB',

                        'RandomForestClassifier',

                        'ExtraTrees',

                        'VotingClassifier'],

                       rotation=40,

                       ha='right')

    plt.title('Training and test score for different classifiers')

    plt.legend([p1[0], p2[0]], ['training', 'test'], loc='lower left')

    plt.show()


vclas(1,3,5,4)
```

You will observe the following output:
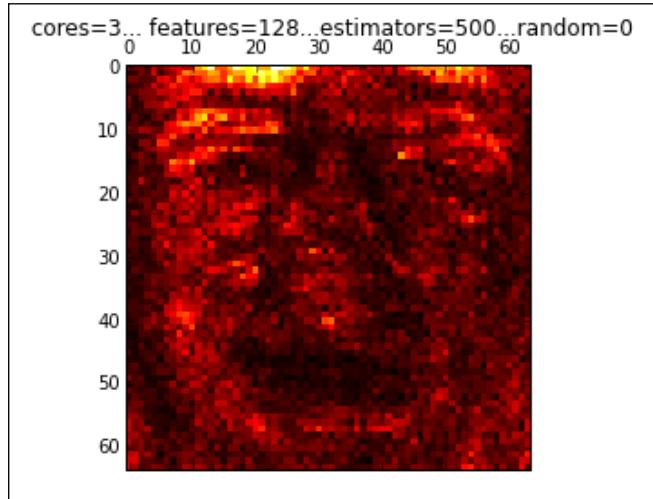
Tree models allow us to assess the relative rank of features in terms of the expected fraction of samples they contribute to. Here, we use one to evaluate the importance of each features in a classification task. A feature's relative importance is based on where it is represented in the tree. Features at the top of a tree contribute to the final decision of a larger proportion of input samples.

The following example uses an `ExtraTreesClassifier` class to map feature importance. The dataset we are using consists of 10 images, each of 40 people, which is 400 images in total. Each image has a label indicating the person's identity. In this task, each pixel is a feature; in the output, the pixel's brightness represents the feature's relative importance. The brighter the pixel, the more important the features. Note that in this model, the brightest pixels are in the forehead region and we should be careful how we interpret this. Since most photographs are illuminated from above the head, the apparently high importance of these pixels may be due to the fact that foreheads tend to be better illuminated, and therefore reveal more detail about an individual, rather than the intrinsic properties of a person's forehead in indicating their identity:

```python
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_olivetti_faces

from sklearn.ensemble import ExtraTreesClassifier

data = fetch_olivetti_faces()

def importance(n_estimators=500, max_features=128, n_jobs=3, random_state=0):

    X = data.images.reshape((len(data.images), -1))

    y = data.target

    forest = ExtraTreesClassifier(n_estimators,max_features=max_features, n_jobs=n_jobs, random_state=random_state)

    forest.fit(X, y)

    dstring=" cores=%d..." % n_jobs + " features=%s..." % max_features +"estimators=%d..." %n_estimators + "random=%d" %random_state

    print(dstring)

    importances = forest.feature_importances_

    importances = importances.reshape(data.images[0].shape)

    plt.matshow(importances, cmap=plt.cm.hot)

    plt.title(dstring)

    #plt.savefig('etreesImportance'+ dstring + '.png')

    plt.show()


importance()
```

The output for the preceding code is as follows:

Boosting

Earlier in this book, I introduced the idea of the PAC learning model and the idea of concept classes. A related idea is that of **weak learnability**. Here each of the learning algorithms in the ensemble need only perform slightly better than chance. For example if each algorithm in the ensemble is correct at least 51% of the time then the criteria of weak learnability are satisfied. It turns out that the ideas of PAC and weak learnability are essentially the same except that for the latter, we drop the requirement that the algorithm must achieve arbitrarily high accuracy. However, it merely performs better than a random hypothesis. How is this useful, you may ask? It is often easier to find rough *rules of thumb* rather than a highly accurate prediction rule. This weak learning model may only perform slightly better than chance; however, if we *boost* this learner by running it many times on different weighted distributions of the data and by combining these learners, we can, hopefully, build a single prediction rule that performs much better than any of the individual weak learning rules.

Boosting is a simple and powerful idea. It extends bagging by taking into account a model's training error. For example, if we train a linear classifier and find that it misclassified a certain set of instances. If we train a subsequent model on a dataset containing duplicates of these misclassified instances, then we would expect that this newly trained model would perform better on a test set. By including duplicates of misclassified instances in the training set, we are shifting the mean of the data set towards these instances. This forces the learner to focus on the most difficult-to-classify examples. This is achieved in practice by giving misclassified instances higher weight and then modifying the model to take this in to account, for example, in a linear classifier we can calculate the class means by using weighted averages.

Starting from a dataset of uniform weights that sum to one, we run the classifier and will likely misclassify some instances. To boost the weight of these instances, we assign them half the total weight. For example, consider a classifier that gives us the following results:

	Predicted positive	Predicted negative	Total
Actual pos.	24	16	40
Actual neg.	9	51	60
Totals	33	67	100

The error rate is $\square = (9 + 16)/100 = 0.25$.

We want to assign half the error weight to the misclassified samples, and since we started with uniform weights that sum to 1, the current weight assigned to the misclassified examples is simply the error rate. To update the weights, therefore, we multiply them by the factor $1/2\square$. Assuming that the error rate is less than 0.5, this results in an increase in the weights of the misclassified examples. To ensure that the weights still sum to 1, we multiply the correctly classified examples by $\frac{1}{2}(1-\square)$. In this example, the error rate, the initial weights of the incorrectly classified samples, is .25 and we want it to be .5, that is, half the total weights, so we multiply this initial error rate by 2. The weights for the correctly classified instances are $1/2(1-\square) = 2/3$. Taking these weights into account results into the following table:

	Predicted positive	Predicted negative	Total
Actual pos.	16	32	48
Actual neg.	18	34	60
Total	33	67	100

The final piece we need is a confidence factor, *a*, which is applied to each model in the ensemble. This is used to make an ensemble prediction based on the weighted averages from each individual model. We want this to increase with decreasing errors. A common way to ensure this happens is to set the confidence factor to the following:

$$\alpha_t = ln \sqrt{\left(\frac{(1-\epsilon_t)}{\epsilon_t}\right)}$$

So we are given a dataset, such as following:

$$(x_1 y_1),\ldots,(x_m y_m) \, where \, x_i \in X, y_i \in Y = \{-1,+1\}$$

We then initialize an equal weighted distribution, such as the following:

$$W_1(i) = \frac{1}{m}$$

Using a weak classifier, h_t, we can write an updated rule as follows:

$$W_{(t+1)}(i) = \frac{\left(W_t(i)exp\left(-\alpha_t y_i h_t(x_i)\right)\right)}{Z_t}$$

With the normalization factor, such as the following:

$$Z_t = \sum_{i=1}^{m}\left(W_t(i)exp\left(-\alpha_t y_i h_t(x_i)\right)\right)$$

Note that $exp(-y_i h_t(x_i))$ is positive and greater than 1 if $-y_i h_t(x_i)$ is positive, and this happens if x_i is misclassified. The result is that the update rule will increase the weight of a misclassified example and decrease the weight of correctly classified samples.

We can write the final classifier as follows:

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha h_t(x)\right)$$

Adaboost

One of the most popular boosting algorithms is called **AdaBoost** or **adaptive boosting**. Here, a decision tree classifier is used as the base learner and it builds a decision boundary on data that is not linearly separable:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_blobs


plot_colors = "br"
plot_step = 0.02
class_names = "AB"
tree= DecisionTreeClassifier()
boost=AdaBoostClassifier()
X,y=make_blobs(n_samples=500,centers=2, random_state=0, cluster_std=2)
boost.fit(X,y)
plt.figure(figsize=(10, 5))


# Plot the decision boundaries
plt.subplot(121)
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                     np.arange(y_min, y_max, plot_step))


Z = boost.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
```
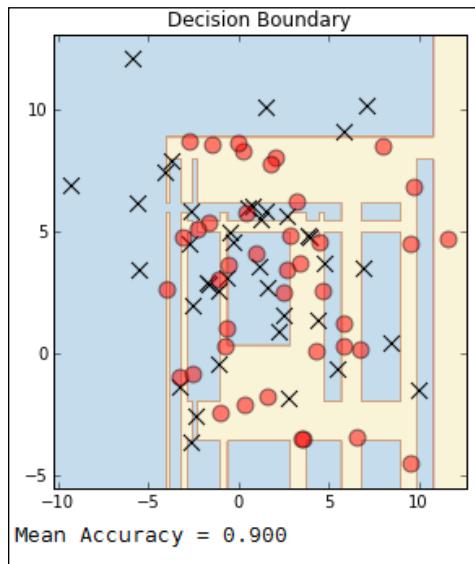
```python
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.axis("tight")


for i, n, c in zip(range(2), class_names, plot_colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1],
                c=c, cmap=plt.cm.Paired,
                label="Class %s" % n)
plt.title('Decision Boundary')


twoclass_output = boost.decision_function(X)
plot_range = (twoclass_output.min(), twoclass_output.max())
plt.subplot(122)
for i, n, c in zip(range(2), class_names, plot_colors):
    plt.hist(twoclass_output[y == i],
             bins=20,
             range=plot_range,
             facecolor=c,
             label='Class %s' % n,
             alpha=.5)
x1, x2, y1, y2 = plt.axis()
plt.axis((x1, x2, y1, y2))
plt.legend(loc='upper left')
plt.ylabel('Samples')
plt.xlabel('Score')
plt.title('Decision Scores')
plt.show()
print("Mean Accuracy =%f" % boost.score(X,y))
```

The following is the output of the preceding commands:

Gradient boosting

Gradient tree boosting is a very useful algorithm for both regression and classification problems. One of its major advantages is that it naturally handles mixed data types, and it is also quite robust to outliers. Additionally, it has better predictive powers than many other algorithms; however, its sequential architecture makes it unsuitable for parallel techniques, and therefore, it does not scale well to large data sets. For datasets with a large number of classes, it is recommended to use `RandomForestClassifier` instead. Gradient boosting typically uses decision trees to build a prediction model based on an ensemble of weak learners, applying an optimization algorithm on the cost function.

In the following example, we create a function that builds a gradient boosting classifier and graphs its cumulative loss versus the number of iterations. The `GradientBoostingClassifier` class has an `oob_improvement_` attribute and is used here calculate an estimate of the test loss on each iteration. This gives us a reduction in the loss compared to the previous iteration. This can be a very useful heuristic for determining the number of optimum iterations. Here, we plot the cumulative improvement of two gradient boosting classifiers. Each classifier is identical but for a different learning rate, *.01* in the case of the dotted line and *.001* for the solid line.

The learning rate shrinks the contribution of each tree, and this means that there is a tradeoff with the number of estimators. Here, we actually see that with a larger learning rate, the model appears to reach its optimum performance faster than the model with a lower learning rate. However, this models appears to achieve better results overall. What usually occurs in practice is that `oob_improvement` deviates in a pessimistic way over a large number of iterations. Let's take a look at the following commands:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import ensemble
from sklearn.cross_validation import train_test_split
from sklearn import datasets


def gbt(params, X,y,ls):
    clf = ensemble.GradientBoostingClassifier(**params)
    clf.fit(X_train, y_train)
    cumsum = np.cumsum(clf.oob_improvement_)
    n = np.arange(params['n_estimators'])
    oob_best_iter = n[np.argmax(cumsum)]
    plt.xlabel('Iterations')
    plt.ylabel('Improvement')
    plt.axvline(x=oob_best_iter,linestyle=ls)
    plt.plot(n, cumsum, linestyle=ls)


X,y=datasets.make_blobs(n_samples=50,centers=5, random_state=0, cluster_
std=5)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
random_state=9)


p1 = {'n_estimators': 1200, 'max_depth': 3, 'subsample': 0.5,
        'learning_rate': 0.01, 'min_samples_leaf': 1, 'random_state':
3}
p2 = {'n_estimators': 1200, 'max_depth': 3, 'subsample': 0.5,
        'learning_rate': 0.001, 'min_samples_leaf': 1, 'random_state':
3}


gbt(p1, X,y, ls='--')
gbt(p2, X,y, ls='-')
```
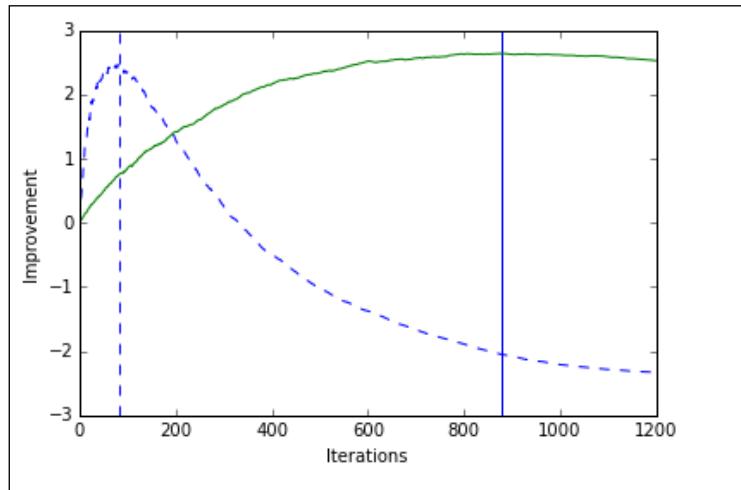
You will observe the following output:

Ensemble strategies

We looked at two broad ensemble techniques: bagging, as applied random forests and extra trees, and boosting, in particular AdaBoost and gradient tree boosting. There are of course many other variants and combinations of these. In the last section of this chapter, I want to examine some strategies for choosing and applying different ensembles to particular tasks.

Generally, in classification tasks, there are three reasons why a model may misclassify a test instance. Firstly, it may simply be unavoidable if features from different classes are described by the same feature vectors. In probabilistic models, this happens when the class distributions overlap so that an instance has non-zero likelihoods for several classes. Here we can only approximate a target hypothesis.

The second reason for classification errors is that the model does not have the expressive capabilities to fully represent the target hypothesis. For example, even the best linear classifier will misclassify instances if the data is not linearly separable. This is due to the bias of the classifier. Although there is no single agreed way to measure bias, we can see that a nonlinear decision boundary will have less bias than a linear one, or that more complex decision boundaries will have less bias than simpler ones. We can also see that tree models have the least bias because they can continue to branch until only a single instance is covered by each leaf.

Now, it may seem that we should attempt to minimize bias; however, in most cases, lowering the bias tends to increase the variance and vice versa. Variance, as you have probably guessed, is the third source of classification errors. High variance models are highly dependent on training data. The nearest neighbor's classifier, for example, segments the instance space into single training points. If a training point near the decision boundary is moved, then that boundary will change. Tree models are also high variance, but for a different reason. Consider that we change the training data in such a way that a different feature is selected at the root of the tree. This will likely result in the rest of the tree being different.

A bagged ensemble of linear classifiers is able to learn a more complicated decision boundary through piecewise construction. Each classifier in the ensemble creates a segment of the decision boundary. This shows that bagging, indeed any ensemble method, is capable of reducing the bias of high bias models. However, what we find in practice is that boosting is generally a more effective way of reducing bias.

> Bagging is primarily a variance reduction technique and boosting is primarily a bias reduction technique.

Bagging ensembles work most effectively with high variance models, such as complex trees, whereas boosting is typically used with high bias models such as linear classifiers.

We can look at boosting in terms of the **margin**. This can be understood as being the signed distance from the decision boundary; a positive sign indicates the correct class and a negative sign a false one. What can be shown is that boosting can increase this margin, even when samples are already on the correct side of the decision boundary. In other words, boosting can continue to improve performance on the test set even when the training error is zero.

Other methods

The major variations on ensemble methods are achieved by changing the way predictions of the base models are combined. We can actually define this as a learning problem in itself, given that the predictions of a set of base classifiers as features learn a **meta-model** that best combines their predictions. Learning a linear meta-model is known as **stacking** or **stacked generalization**. Stacking uses a weighted combination of all learners and, in a classification task, a combiner algorithm such as logistic regression is used to make the final prediction. Unlike bagging or boosting, and like bucketing, stacking is often used with models of different types.

Typical stacking routines involve the following steps:

1. Split the training set into two disjointed sets.
2. Train several base learners on the first set.
3. Test the base learner on the second set.
4. Use the predictions from the previous step to train a higher level learner.

Note that the first three steps are identical to cross validation; however, rather than taking a winner-takes-all approach, the base learners are combined, possibly nonlinearly.

A variation on this theme is **bucketing**. Here, a selection algorithm is used to choose the best model for each problem. This can be done, for example, using a perception to pick the best model by giving a weight to the predictions of each model. With a large set of diverse models, some will take longer to train than others. A way to use this in an ensemble is to first use the fast but imprecise algorithms to choose which slower, but more accurate, algorithms will likely do best.

We can incorporate diversity using a heterogeneous set of base learners. This diversity comes from the different learning algorithms and not the data. This means that each model can use the same training set. Often, the base models consist of sets of the same type but with different hyper parameter settings.

Ensembles, in general, consist of a set of base models and a meta-model that are trained to find the best way to combine these base models. If we are using a weighted set of models and combining their output in some way, we assume that if a model has a weight close to zero, then it will have very little influence on the output. It is conceivable that a base classifier has a negative weight, and in this case, its prediction would be inverted, relative to the other base models. We can even go further and attempt to predict how well a base model is likely to perform even before we train it. This is sometimes called **meta-learning**. This involves, first, training a variety of models on a large collection of data and constructing a model that will help us answer questions such as which model is likely to outperform another model on a particular dataset, or does the data indicate that particular (meta) parameters are likely to work best?

Remember that no learning algorithm can outperform another when evaluated over the space of all possible problems, such as predicting the next number is a sequence if all possible sequences are likely. Of course, learning problems in the real world have nonuniform distributions, and this enables us to build prediction models on them. The important question in meta-learning is how to design the features on which the meta-model is built. They need to combine the relevant characteristics of both the trained model and the dataset. This must include aspects of the data beyond the number and type of features, and the number of samples.

Summary

In this chapter, we looked at the major ensemble methods and their implementations in scikit-learn. It is clear that there is a large space to work in and finding what techniques work best for different types of problems is the key challenge. We saw that the problems of bias and variance each have their own solution, and it is essential to understand the key indicators of each of these. Achieving good results usually involves much experimentation, and using some of the simple techniques described in this chapter, you can begin your journey into machine learning ensembles.

In the next and last chapter, we will introduce the most important topic—model selection and evaluation—and examine some real-world problems from different perspectives.

9
Design Strategies and Case Studies

With the possible exception of data **munging**, evaluating is probably what machine learning scientists spend most of their time doing. Staring at lists of numbers and graphs, watching hopefully as their models run, and trying earnestly to make sense of their output. Evaluation is a cyclical process; we run models, evaluate the results, and plug in new parameters, each time hoping that this will result in a performance gain. Our work becomes more enjoyable and productive as we increase the efficiency of each evaluation cycle, and there are some tools and techniques that can help us achieve this. This chapter will introduce some of these through the following topics:

- Evaluating model performance
- Model selection
- Real-world case studies.
- Machine learning design at a glance

Evaluating model performance

Measuring a model's performance is an important machine learning task, and there are many varied parameters and heuristics for doing this. The importance of defining a scoring strategy should not be underestimated, and in Sklearn, there are basically three approaches:

- **Estimator score**: This refers to using the estimator's inbuilt `score()` method, specific to each estimator
- **Scoring parameters**: This refers to cross-validation tools relying on an internal scoring strategy
- **Metric functions**: These are implemented in the metrics module

We have seen examples of the estimator `score()` method, for example, `clf.score()`. In the case of a linear classifier, the `score()` method returns the mean accuracy. It is a quick and easy way to gauge an individual estimator's performance. However, this method is usually insufficient in itself for a number of reasons.

If we remember, accuracy is the sum of the true positive and true negative cases divided by the number of samples. Using this as a measure would indicate that if we performed a test on a number of patients to see if they had a particular disease, simply predicting that every patient was disease free would likely give us a high accuracy. Obviously, this is not what we want.

A better way to measure performance is using by **precision**, (**P**) and **Recall**, (**R**). If you remember from the table in *Chapter 4*, *Models – Learning from Information*, precision, or specificity, is the proportion of predicted positive instances that are correct, that is, *TP/(TP+FP)*. Recall, or sensitivity, is *TP/(TP+FN)*. The F-measure is defined as *2*R*P/(R+P)*. These measures ignore the true negative rate, and so they are not making an evaluation on how well a model handles negative cases.

Rather than use the score method of the estimator, it often makes sense to use specific scoring parameters such as those provided by the `cross_val_score` object. This has a `cv` parameter that controls how the data is split. It is usually set as an int, and it determines how many random consecutive splits are made on the data. Each of these has a different split point. This parameter can also be set to an iterable of train and test splits, or an object that can be used as a cross validation generator.

Also important in `cross_val_score` is the scoring parameter. This is usually set by a string indicating a scoring strategy. For classification, the default is *accuracy*, and some common values are `f1`, `precision`, `recall`, as well as the micro-averaged, macro-averaged, and weighted versions of these. For regression estimators, the `scoring` values are `mean_absolute_error`, `mean_squared error`, `median_absolute_error`, and `r2`.

The following code estimates the performance of three models on a dataset using 10 consecutive splits. Here, we print out the mean of each score, using several measures, for each of the four models. In a real-world situation, we will probably need to preprocess our data in one or more ways, and it is important to apply these data transformations to our test set as well as the training set. To make this easier, we can use the `sklearn.pipeline` module. This sequentially applies a list of transforms and a final estimator, and it allows us to assemble several steps that can be cross-validated together. Here, we also use the `StandardScaler()` class to scale the data. Scaling is applied to the logistic regression model and the decision tree by using two pipelines:

```
from sklearn import cross_validation
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn import svm

from sklearn.linear_model import LogisticRegression

from sklearn.datasets import samples_generator

from sklearn.preprocessing import LabelEncoder

from sklearn.preprocessing import StandardScaler

from sklearn.cross_validation import cross_val_score

from sklearn.pipeline import Pipeline

X, y = samples_generator.make_classification(n_samples=1000,n_
informative=5, n_redundant=0,random_state=42)

le=LabelEncoder()

y=le.fit_transform(y)

Xtrain, Xtest, ytrain, ytest = cross_validation.train_test_split(X, y,
test_size=0.5, random_state=1)

clf1=DecisionTreeClassifier(max_depth=2,criterion='gini').
fit(Xtrain,ytrain)

clf2= svm.SVC(kernel='linear', probability=True, random_state=0).
fit(Xtrain,ytrain)

clf3=LogisticRegression(penalty='l2', C=0.001).fit(Xtrain,ytrain)

pipe1=Pipeline([['sc',StandardScaler()],['mod',clf1]])

mod_labels=['Decision Tree','SVM','Logistic Regression' ]

print('10 fold cross validation: \n')

for mod,label in zip([pipe1,clf2,clf3], mod_labels):

    #print(label)

    auc_scores= cross_val_score(estimator= mod, X=Xtrain, y=ytrain,
cv=10, scoring ='roc_auc')

    p_scores= cross_val_score(estimator= mod, X=Xtrain, y=ytrain, cv=10,
scoring ='precision_macro')

    r_scores= cross_val_score(estimator= mod, X=Xtrain, y=ytrain, cv=10,
scoring ='recall_macro')

    f_scores= cross_val_score(estimator= mod, X=Xtrain, y=ytrain, cv=10,
scoring ='f1_macro')


    print(label)

    print("auc scores %2f +/- %2f " % (auc_scores.mean(), auc_scores.
std()))

    print("precision %2f +/- %2f " % (p_scores.mean(), p_scores.std()))

    print("recall %2f +/- %2f ]" % (r_scores.mean(), r_scores.std()))

    print("f scores %2f +/- %2f " % (f_scores.mean(), f_scores.std()))
```

On execution, you will see the following output:

```
10 fold cross validation:

Decision Tree
auc scores 0.692144 +/- 0.056865
precision 0.706912 +/- 0.065688
recall 0.648131 +/- 0.043604 ]
f scores 0.628455 +/- 0.051711
SVM
auc scores 0.768374 +/- 0.038460
precision 0.709994 +/- 0.058011
recall 0.707064 +/- 0.056323 ]
f scores 0.703605 +/- 0.055579
Logistic Regression
auc scores 0.754150 +/- 0.048137
precision 0.688979 +/- 0.077614
recall 0.686077 +/- 0.076052 ]
f scores 0.682859 +/- 0.075356
```

There are several variations on these techniques, most commonly using what is known as **k-fold cross validation**. This uses what is sometimes referred to as the *leave one out* strategy. First, the model is trained using $k-1$ of the folds as training data. The remaining data is then used to compute the performance measure. This is repeated for each of the folds. The performance is calculated as an average of all the folds.

Sklearn implements this using the `cross_validation.KFold` object. The important parameters are a required `int`, indicating the total number of elements, and an `n_folds` parameter, defaulting to `3`, to indicate the number of folds. It also takes optional `shuffle` and `random_state` parameters indicating whether to shuffle the data before splitting, and what method to use to generate the random state. The default `random_state` parameter is to use the NumPy random number generator.

In the following snippet, we use the `LassoCV` object. This is a linear model trained with L1 regularization. The optimization function for regularized linear regression, if you remember, includes a constant (alpha) that multiplies the L1 regularization term. The `LassoCV` object automatically sets this alpha value, and to see how effective this is, we can compare the selected alpha and the score on each of the k-folds:

```
import numpy as np

from sklearn import cross_validation, datasets, linear_model

X,y=datasets.make_blobs(n_samples=80,centers=2, random_state=0, cluster_
std=2)

alphas = np.logspace(-4, -.5, 30)

lasso_cv = linear_model.LassoCV(alphas=alphas)
```

```
k_fold = cross_validation.KFold(len(X), 5)
alphas = np.logspace(-4, -.5, 30)


for k, (train, test) in enumerate(k_fold):
    lasso_cv.fit(X[train], y[train])
    print("[fold {0}] alpha: {1:.5f}, score: {2:.5f}".
        format(k, lasso_cv.alpha_, lasso_cv.score(X[test], y[test])))
```

The output of the preceding commands is as follows:

```
[fold 0] alpha: 0.01964, score: 0.42157
[fold 1] alpha: 0.00853, score: 0.52112
[fold 2] alpha: 0.00010, score: 0.48277
[fold 3] alpha: 0.00010, score: 0.42657
[fold 4] alpha: 0.00489, score: 0.54747
```

Sometimes, it is necessary to preserve the percentages of the classes in each fold. This is done using **stratified cross validation**. It can be helpful when classes are unbalanced, that is, when there is a larger number of some classes and very few of others. Using the stratified cv object may help correct defects in models that might cause bias because a class is not represented in a fold in large enough numbers to make an accurate prediction. However, this may also cause an unwanted increase in variance.

In the following example, we use stratified cross validation to test how significant the classification score is. This is done by repeating the classification procedure after randomizing the labels. The *p* value is the percentage of runs by which the score is greater than the classification score obtained initially. This code snippet uses the cross_validation.permutation_test_score method that takes the estimator, data, and labels as parameters. Here, we print out the initial test score, the *p* value, and the score on each permutation:

```
import numpy as np
from sklearn import linear_model
from sklearn.cross_validation import StratifiedKFold, permutation_test_
score
from sklearn import datasets


X,y=datasets.make_classification(n_samples=100, n_features=5)
n_classes = np.unique(y).size
cls=linear_model.LogisticRegression()
```

```
cv = StratifiedKFold(y, 2)
score, permutation_scores, pvalue = permutation_test_score(cls, X, y,
scoring="f1", cv=cv, n_permutations=10, n_jobs=1)


print("Classification score %s (pvalue : %s)" % (score, pvalue))
print("Permutation scores %s" % (permutation_scores))
```

This gives the following output:

```
Classification score 0.968962585034 (pvalue : 0.0909090909091)
Permutation scores [ 0.36310273  0.57189542  0.55977011  0.38134058  0.50802139  0.47916667
  0.47153537  0.3797519   0.46071429  0.49      ]
```

Model selection

There are a number of hyper parameters that can be adjusted to improve performance. It is often not a straightforward process, determining the effect of the various parameters, both individually and in combination with each other. Common things to try include getting more training examples, adding or removing features, adding polynomial features, and increasing or decreasing the regularization parameter. Given that we can spend a considerable amount of time collecting more data, or manipulating data in other ways, it is important that the time you spend is likely to result in a productive outcome. One of the most important ways to do this is using a process known as grid search.

Gridsearch

The `sklearn.grid_search.GridSearchCV` object is used to perform an exhaustive search on specified parameter values. This allows iteration through defined sets of parameters and the reporting of the result in the form of various metrics. The important parameters for `GridSearchCV` objects are an estimator and a parameter grid. The `param_grid` parameter is a dictionary, or list of dictionaries, with parameter names as keys and a list of parameter settings to try as values. This enables searching over any sequence of the estimators parameter values. Any of an estimator's adjustable parameters can be used with grid search. By default, grid search uses the `score()` function of the estimator to evaluate a parameter value. For classification, this is the accuracy, and as we have seen, this may not be the best measure. In this example, we set the scoring parameter of the `GridSearchCV` object to `f1`.

In the following code, we perform a search over a range of `C` values (the inverse regularization parameter), under both L1 and L2 regularization. We use the `metrics. classification_report` class to print out a detailed classification report:

```python
from sklearn import datasets
from sklearn.cross_validation import train_test_split
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.linear_model import LogisticRegression as lr


X,y=datasets.make_blobs(n_samples=800,centers=2, random_state=0, cluster_
std=4)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.5, random_state=0)
tuned_parameters = [{'penalty': ['l1'],
                     'C': [0.01, 0.1, 1, 5]},
                    {'penalty': ['l2'], 'C': [0.01, 0.1, 1, 5]}]
scores = ['precision', 'recall','f1']
for score in scores:
    clf = GridSearchCV(lr(C=1), tuned_parameters, cv=5,
                       scoring='%s_weighted' % score)
    clf.fit(X_train, y_train)
    print("Best parameters on development set:")
    print()
    print(clf.best_params_)
    print("Grid scores on development set:")
    for params, mean_score, scores in clf.grid_scores_:
        print("%0.3f (+/-%0.03f) for %r"
              % (mean_score, scores.std() * 2, params))
    print("classification report:")
    y_true, y_pred = y_test, clf.predict(X_test)
    print(classification_report(y_true, y_pred))
```

We observe the following output:

```
Best parameters on development set:
{'penalty': 'l1', 'C': 0.1}
Grid scores on development set:
0.680 (+/-0.069) for {'penalty': 'l1', 'C': 0.01}
0.707 (+/-0.121) for {'penalty': 'l1', 'C': 0.1}
0.695 (+/-0.122) for {'penalty': 'l1', 'C': 1}
0.699 (+/-0.128) for {'penalty': 'l1', 'C': 5}
0.706 (+/-0.111) for {'penalty': 'l2', 'C': 0.01}
0.697 (+/-0.112) for {'penalty': 'l2', 'C': 0.1}
0.702 (+/-0.132) for {'penalty': 'l2', 'C': 1}
0.702 (+/-0.132) for {'penalty': 'l2', 'C': 5}
classification report:
              precision    recall  f1-score   support

           0       0.62      0.77      0.69       189
           1       0.73      0.58      0.65       211

avg / total       0.68      0.67      0.67       400
```

Grid search is probably the most used method of optimization hyper parameters, however, there are times when it may not be the best choice. The `RandomizedSearchCV` object implements a randomized search over possible parameters. It uses a dictionary similar to the `GridSearchCV` object, however, for each parameter, a distribution can be set, over which a random search of values will be made. If the dictionary contains a list of values, then these will be sampled uniformly. Additionally, the `RandomizedSearchCV` object also contains an `n_iter` parameter that is effectively a computational budget of the number of parameter settings sampled. It defaults to 10, and at high values, will generally give better results. However, this is at the expense of runtime.

There are alternatives to the *brute force* approach of the grid search, and these are provided in estimators such as `LassoCV` and `ElasticNetCV`. Here, the estimator itself optimizes its regularization parameter by fitting it along a regularization, *path*. This is usually more efficient than using a grid search.

Learning curves

An important way to understand how a model is performing is by using learning curves. Consider what happens to the training and test errors as we increase the number of samples. Consider a simple linear model. With few training samples, it is very easy for it to fit the parameters, the training error will be small. As the training set grows, it becomes harder to fit, and the average training error will likely grow. On the other hand, the cross validation error will likely decrease, at least at the beginning, as samples are added. With more samples to train on, the model will be better able to acclimatize to new samples. Consider a model with high bias, for example, a simple linear classifier with two parameters. This is just a straight line, so as we start adding training examples, the cross validation error will initially decrease. However, after a certain point, adding training examples will not reduce the error significantly simply because of the limitations of a straight line, it simply cannot fit nonlinear data. If we look at the training error, we see that, like earlier, it initially increases with more training samples, and at a certain point, it will approximately equal the cross validation error. Both the cross validation and train errors will be high in a high-bias example. What this shows is that if we know our learning algorithm has high bias, then just adding more training examples will be unlikely to improve the model significantly.

Now, consider a model with high variance, say with a large number of polynomial terms, and a small value for the regularization parameter. As we add more samples, the training error will increase slowly but remain relatively small. As more training samples are added the error on the cross validation set will decrease. This is an indication of over-fitting. The indicative characteristic of a model with high variance is a large difference between the training error and the test error. What this is showing is that increasing training examples will lower the cross validation error, and therefore, adding training samples is a likely way to improve a model with high variance.

In the following code, we use the learning curve object to plot the test error and the training error as we increase the sample size. This should give you an indication when a particular model is suffering from high bias or high variance. In this case, we are using a logistic regression model. We can see from the output of this code that the model may be suffering from bias, since both the training test errors are relatively high:

```
from sklearn.pipeline import Pipeline

from sklearn.learning_curve import learning_curve

import matplotlib.pyplot as plt

import numpy as np

from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import LogisticRegression
```

```
from sklearn import cross_validation
from sklearn import datasets


X, y = datasets.make_classification(n_samples=2000,n_informative=2, n_
redundant=0,random_state=42)
Xtrain, Xtest, ytrain, ytest = cross_validation.train_test_split(X, y,
test_size=0.5, random_state=1)
pipe = Pipeline ([('sc' , StandardScaler()),('clf', LogisticRegression(
penalty = 'l2'))])
trainSizes, trainScores, testScores = learning_curve(estimator=pipe,
X=Xtrain, y= ytrain,train_sizes=np.linspace(0.1,1,10),cv=10, n_jobs=1)
trainMeanErr=1-np.mean(trainScores, axis=1)
testMeanErr=1-np.mean(testScores, axis=1)
plt.plot(trainSizes, trainMeanErr, color='red', marker='o', markersize=5,
label = 'training error')
plt.plot(trainSizes, testMeanErr, color='green', marker='s',
markersize=5, label = 'test error')
plt.grid()
plt.xlabel('Number of Training Samples')
plt.ylabel('Error')
plt.legend(loc=0)
plt.show()
```

Here is the output of the preceding code:

Real-world case studies

Now, we will move on to some real-world machine learning scenarios. First, we will build a recommender system, and then we will look into some integrated pest management systems in greenhouses.

Building a recommender system

Recommender systems are a type of information filtering, and there are two general approaches: **content-based filtering** and **collaborative filtering**. In content-based filtering, the system attempts to model a user's long term interests and select items based on this. On the other hand, collaborative filtering chooses items based on the correlation with items chosen by people with similar preferences. As you would expect, many systems use a hybrid of these two approaches.

Content-based filtering

Content-based filtering uses the content of items, which is represented as a set of descriptor terms, and matches them with a user profile. A user profile is constructed using the same terms extracted from items that the user has previously viewed. A typical online book store will extract key terms from texts to create a user profile and to make recommendations. This procedure of extracting these terms can be automated in many cases, although in situations where specific domain knowledge is required, these terms may need to be added manually. The manual addition of terms is particularly relevant when dealing with non-text based items. It is relatively easy to extract key terms from, say, a library of books, say by associating fender amplifiers with electric guitars. In many cases, this will involve a human creating these associations based on specific domain knowledge, say by associating fender amplifiers with electric guitars. Once this is constructed, we need to choose a learning algorithm that can learn a user profile and make appropriate recommendations. The two models that are most often used are the vector space model and the latent semantic indexing model. With the vector space model, we create a sparse vector representing a document where each distinct term in a document corresponds to a dimension of the vector. Weights are used to indicate whether a term appears in a document. When it does appear, it shows the weight of 1, and when it does not, it shows the weight of 0. Weights based on the number of times a word appears are also used.

The alternative model, latent semantic indexing, can improve the vector model in several ways. Consider the fact that the same concept is often described by many different words, that is, with synonyms. For example, we need to know that a computer monitor and computer screen are, for most purposes, the same thing. Also, consider that many words have more than one distinct meaning, for example, the word *mouse* can either be an animal or a computer interface. Semantic indexing incorporates this information by building a **term-document** matrix. Each entry represents the number of occurrences of a particular term in the document. There is one row for each of the terms in a set of documents, and there is one column for every document. Through a mathematical process known as single value decomposition this single matrix can be decomposed into three matrices representing documents and terms as vectors of factor values. Essentially this is a dimension reduction technique whereby we create single features that represent multiple words. A recommendation is made based on these derived features. This recommendation is based on semantic relationships within the document rather than simply matching on identical words. The disadvantages of this technique is that it is computationally expensive and may be slow to run. This can be a significant constraint for a recommender system that has to work in realtime.

Collaborative filtering

Collaborative filtering takes a different approach and is used in a variety of settings, particularly, in the context of social media, and there are a variety of ways to implement it. Most take a *neighborhood* approach. This is based on the idea that you are more likely to trust your friends' recommendations, or those with similar interests, rather than people you have less in common with.

In this approach, a weighted average of the recommendations of other people is used. The weights are determined by the correlation between individuals. That is, those with similar preferences will be weighted higher than those that are less similar. In a large system with many thousands of users, it becomes infeasible to calculate all the weights at runtime. Instead, the recommendations of a *neighborhood* are used. This neighborhood is selected either by using a certain weight threshold, or by selecting based on the highest correlation.

n the following code, we use a dictionary of users and their ratings of music albums. The geometric nature of this model is most apparent when we plot users' ratings of two albums. It is easy to see that the distance between users on the plot is a good indication of how similar their ratings are. The Euclidean distance measures how far apart users are, in terms of how closely their preferences match. We also need a way to take into account associations between two people, and for this we use the Pearson correlation index. Once we can compute the similarity between users, we rank them in order of similarity. From here, we can work out what albums could be recommended. This is done by multiplying the similarity score of each user by their ratings. This is then summed and divided by the similarity score, essentially calculating a weighted average based on the similarity score.

Another approach is to find the similarities between items. This is called **item-based collaborative filtering**; this in contrast with user-based collaborative filtering, which we used to calculate the similarity score. The item-based approach is to find similar items for each item. Once we have the similarities between all the albums, we can generate recommendations for a particular user.

Let's take a look at a sample code implementation:

```python
import pandas as pd
from scipy.stats import pearsonr
import matplotlib.pyplot as plt


userRatings={'Dave': {'Dark Side of Moon': 9.0,
  'Hard Road': 6.5,'Symphony 5': 8.0,'Blood Cells': 4.0},'Jen': {'Hard
Road': 7.0,'Symphony 5': 4.5,'Abbey Road':8.5,'Ziggy Stardust': 9,'Best
Of Miles':7},'Roy': {'Dark Side of Moon': 7.0,'Hard Road': 3.5,'Blood
Cells': 4,'Vitalogy': 6.0,'Ziggy Stardust': 8,'Legend': 7.0,'Abbey
Road': 4},'Rob': {'Mass in B minor': 10,'Symphony 5': 9.5,'Blood Cells':
3.5,'Ziggy Stardust': 8,'Black Star': 9.5,'Abbey Road': 7.5},'Sam':
{'Hard Road': 8.5,'Vitalogy': 5.0,'Legend': 8.0,'Ziggy Stardust':
9.5,'U2 Live': 7.5,'Legend': 9.0,'Abbey Road': 2},'Tom': {'Symphony 5':
4,'U2 Live': 7.5,'Vitalogy': 7.0,'Abbey Road': 4.5},'Kate': {'Horses':
8.0,'Symphony 5': 6.5,'Ziggy Stardust': 8.5,'Hard Road': 6.0,'Legend':
8.0,'Blood Cells': 9,'Abbey Road': 6}}


# Returns a distance-based similarity score for user1 and user2
def distance(prefs,user1,user2):
    # Get the list of shared_items
    si={}
```

```
        for item in prefs[user1]:
            if item in prefs[user2]:
                si[item]=1
    # if they have no ratings in common, return 0
    if len(si)==0: return 0
    # Add up the squares of all the differences
    sum_of_squares=sum([pow(prefs[user1][item]-prefs[user2][item],2)
    for item in prefs[user1] if item in prefs[user2]])
    return 1/(1+sum_of_squares)


def Matches(prefs,person,n=5,similarity=pearsonr):
    scores=[(similarity(prefs,person,other),other)
        for other in prefs if other!=person]
    scores.sort( )
    scores.reverse( )
    return scores[0:n]


def getRecommendations(prefs,person,similarity=pearsonr):
    totals={}
    simSums={}
    for other in prefs:
        if other==person: continue
        sim=similarity(prefs,person,other)
        if sim<=0: continue
        for item in prefs[other]:
            # only score albums not yet rated
            if item not in prefs[person] or prefs[person][item]==0:
                # Similarity * Score
                totals.setdefault(item,0)
                totals[item]+=prefs[other][item]*sim
                # Sum of similarities
                simSums.setdefault(item,0)
                simSums[item]+=sim
    # Create a normalized list
    rankings=[(total/simSums[item],item) for item,total in totals.items(
)]
    # Return a sorted list
    rankings.sort( )
```

```
    rankings.reverse( )
    return rankings


def transformPrefs(prefs):
    result={}
    for person in prefs:
        for item in prefs[person]:
            result.setdefault(item,{})
            # Flip item and person
            result[item][person]=prefs[person][item]
    return result


transformPrefs(userRatings)


def calculateSimilarItems(prefs,n=10):
    # Create a dictionary similar items
    result={}
    # Invert the preference matrix to be item-centric
    itemPrefs=transformPrefs(prefs)
    for item in itemPrefs:
#        if c%100==0: print("%d / %d" % (c,len(itemPrefs)))
        scores=Matches(itemPrefs,item,n=n,similarity=distance)
        result[item]=scores
    return result


def getRecommendedItems(prefs,itemMatch,user):
    userRatings=prefs[user]
    scores={}
    totalSim={}

    # Loop over items rated by this user
    for (item,rating) in userRatings.items( ):

        # Loop over items similar to this one
        for (similarity,item2) in itemMatch[item]:

            # Ignore if this user has already rated this item
            if item2 in userRatings: continue
```

```
            # Weighted sum of rating times similarity
            scores.setdefault(item2,0)
            scores[item2]+=similarity*rating

            # Sum of all the similarities
            totalSim.setdefault(item2,0)
            totalSim[item2]+=similarity


    # Divide each total score by total weighting to get an average
    rankings=[(score/totalSim[item],item) for item,score in scores.items(
)]


    # Return the rankings from highest to lowest
    rankings.sort( )
    rankings.reverse( )
    return rankings


itemsim=calculateSimilarItems(userRatings)


def plotDistance(album1, album2):
    data=[]
    for i in userRatings.keys():
        try:
            data.append((i,userRatings[i][album1], userRatings[i]
[album2]))
        except:
            pass
    df=pd.DataFrame(data=data, columns = ['user', album1, album2])
    plt.scatter(df[album1],df[album2])
    plt.xlabel(album1)
    plt.ylabel(album2)
    for i,t in enumerate(df.user):
        plt.annotate(t,(df[album1][i], df[album2][i]))
    plt.show()
    print(df)


plotDistance('Abbey Road', 'Ziggy Stardust')
print(getRecommendedItems(userRatings, itemsim,'Dave'))
```
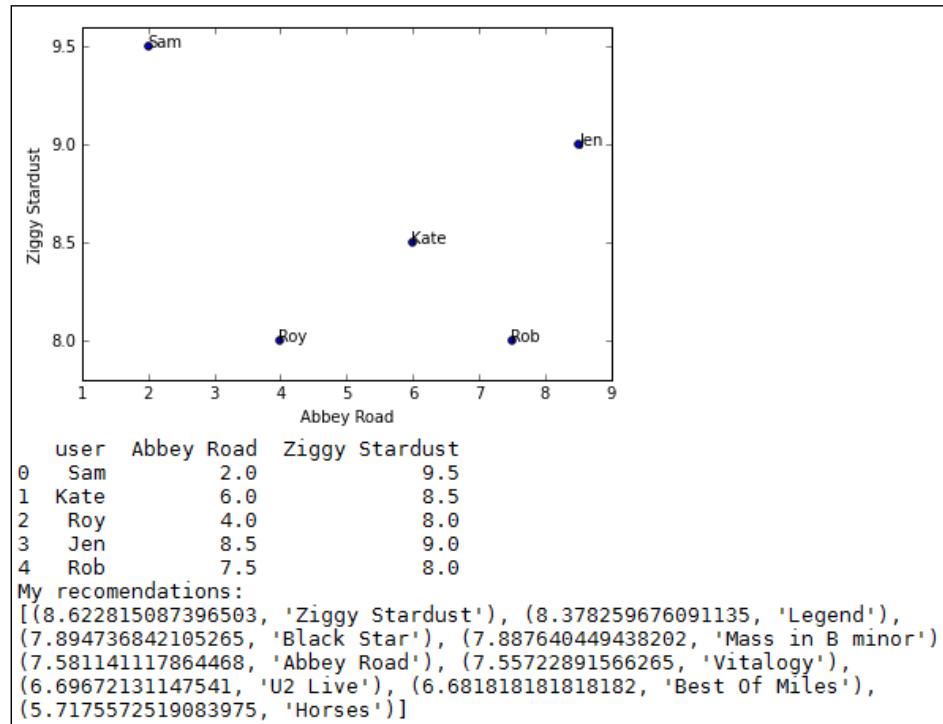
You will observe the following output:


```
   user  Abbey Road  Ziggy Stardust
0   Sam         2.0             9.5
1  Kate         6.0             8.5
2   Roy         4.0             8.0
3   Jen         8.5             9.0
4   Rob         7.5             8.0
My recomendations:
[(8.622815087396503, 'Ziggy Stardust'), (8.378259676091135, 'Legend'),
(7.894736842105265, 'Black Star'), (7.887640449438202, 'Mass in B minor'),
(7.581141117864468, 'Abbey Road'), (7.55722891566265, 'Vitalogy'),
(6.69672131147541, 'U2 Live'), (6.681818181818182, 'Best Of Miles'),
(5.7175572519083975, 'Horses')]
```

Here we have plotted the user ratings of two albums, and based on this, we can see that the users **Kate** and **Rob** are relatively close, that is, their preferences with regard to these two albums are similar. On the other hand, the users **Rob** and **Sam** are far apart, indicating different preferences for these two albums. We also print out recommendations for the user **Dave** and the similarity score for each album recommended.

Since collaborative filtering is reliant on the ratings of other users, a problem arises when the number of documents becomes much larger than the number of ratings, so the number of items that a user has rated is a tiny proportion of all the items. There are a few different approaches to help you fix this. Ratings can be inferred from the type of items they browse for on the site. Another way is to supplement the ratings of users with content-based filtering in a hybrid approach.

Reviewing the case study

Some important aspects of this case study are as follows:

- It is part of a web application. It must run in realtime, and it relies on user interactivity.

- There are extensive practical and theoretical resources available. This is a well thought out problem and has several well defined solutions. We do not have to reinvent the wheel.

- This is largely a marketing project. It has a quantifiable metric of success in that of sale volumes based on recommendation.

- The cost of failure is relatively low. A small level of error is acceptable.
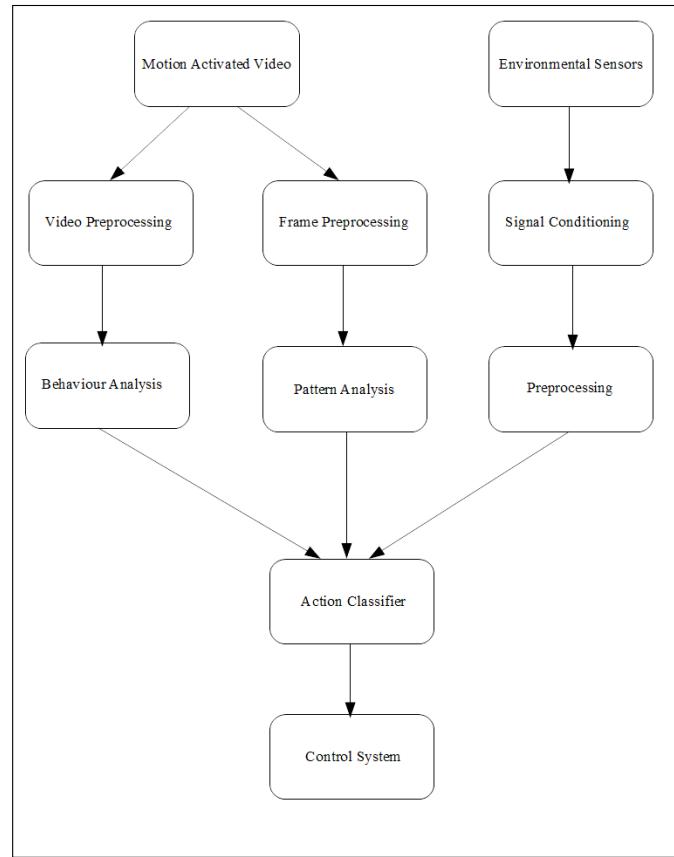
Insect detection in greenhouses

A growing population and increasing climate variability pose unique challenges for agriculture in the 21st century. The ability of controlled environments, such as greenhouses, to provide optimum growing conditions and maximize the efficient use of inputs, such as water and nutrients, will enable us to continue to feed growing populations in a changing global climate.

There are many food production systems that today are largely automated, and these can be quite sophisticated. Aquaculture systems can cycle nutrients and water between fish tanks and growing racks, in essence, creating a very simple ecology in an artificial environment. The nutrient content of the water is regulated, as are the temperature, moisture levels, humidity, and carbon dioxide levels. These features exist within very precise ranges to optimize for production.

The environmental conditions inside greenhouses can be very conducive to the rapid spread of disease and pests. Early detection and the detection of precursor symptoms, such as fungi or insect egg production, are essential to managing these diseases and pests. For environmental, food quality, and economic reasons, we want to only apply minimum targeted controls, since this mostly involves the application, a pesticide, or any other bio agent.

The goal here is to create an automated system that will detect the type and location of a disease or insect and subsequently choose, and ideally implement, a control. This is quite a large undertaking with a number of different components. Many of the technologies exist separately, but here we are combining them in a number of non-standard ways. The approach is largely experimental:

The usual method of detection has been direct human observation. This is a very time intensive task and requires some particular skills. It is also very error prone. Automating this would be of huge benefit in itself, as well as being an important starting point for creating an automated IPM system. One of the first tasks is to define a set of indicators for each of the targets. A natural approach would be to get an expert, or a panel of experts, to classify short video clips as either being pest free or infected with one or more target species. Next, a classifier is trained on these clips, and hopefully, it is able to obtain a prediction. This approach has been used in the past, for example, *Early Pest Detection in Greenhouses* (Martin, Moisan, 2004), in the detection of insect pests.

In a typical setup, video cameras are placed throughout the greenhouse to maximize the sampling area. For the early detection of pests, key plant organs such as the stems, leaf nodes, and other areas are targeted. Since video and image analysis can be computationally expensive, motion sensitive cameras that are intelligently programmed to begin recording when they detect insect movement can be used.

The changes in early outbreaks are quite subtle and can be indicated to be a combination of plant damage, discolorations, reduced growth, and the presence of insects or their eggs. This difficulty is compounded by the variable light conditions in greenhouses. A way of coping with these issues is to use a cognitive vision approach. This divides the problem into a number of sub-problems, each of which is context dependent. For example, the use a different model for when it is sunny, or based on the light conditions at different times of the day. The knowledge of this context can be built into the model at a preliminary, weak learning stage. This gives it an inbuilt heuristic to apply an appropriate learning algorithm in a given context.

An important requirement is that we distinguish between different insect species, and a way to do this is by capturing the dynamic components of insects, that is, their behavior. Many insects can be distinguished by their type of movement, for example, flying in tight circles, or stationary most of the time with short bursts of flight. Also, insects may have other behaviors, such as mating or laying eggs, that might be an important indicator of a control being required.

Monitoring can occur over a number of channels, most notably video and still photography, as well as using signals from other sensors such as infrared, temperature, and humidity sensors. All these inputs need to be time and location stamped so that they can be used meaningfully in a machine learning model.

Video processing first involves subtracting the background and isolating the moving components of the sequence. At the pixel-level, the lighting condition results in a variation of intensity, saturation, and inter-pixel contrast. At the image level, conditions such as shadows affect only a portion of the image, whereas backlighting affects the entire image.

In this example, we extract frames from the video recordings and process them in their own separate path in the system. As opposed to video processing, where we were interested in the sequence of frames over time in an effort to detect movement, here we are interested in single frames from several cameras, focused on the same location at the same time. This way, we can build up a three-dimensional model, and this can be useful, especially for tracking changes to biomass volume.

The final inputs for our machine learning model are environmental sensors. Standard control systems measure temperature, relative humidity, carbon dioxide levels, and light. In addition, hyper-spectral and multi-spectral sensors are capable of detecting frequencies outside the visible spectrum. The nature of these signals requires their own distinctive processing paths. As an example of how they might be used, consider that one of our targets is a fungus that we know exists in a narrow range of humidity and temperature. Supposing an ultraviolet sensor in a part of the greenhouse briefly detects the frequency range indicative of the fungi. Our model would register this, and if the humidity and temperature are in this range, then a control may be initiated. This control may be simply the opening of a vent or the switching on of a fan near the possible outbreak to locally cool the region to a temperature at which the fungi cannot survive.

Clearly, the most complex part of the system is the action controller. This really comprises of two elements: A multi label classifier outputting a binary vector representing the presence or not of the target pests and the action classifier itself which outputs a control strategy.

There are many different components and a number of distinct systems that are needed to detect the various pathogens and pests. The standard approach has been to create a separate learning model for each target. This multi-model approach works if we are instigating controls for each of these as separate, unrelated activities. However, many of the processes, such as the development and spread of disease and a sudden outbreak of insects, may be precipitated by a common cause.

Reviewing the case study

Some important aspects of this case study are as follows:

- It is largely a research project. It has a long timeline involving a large space of unknowns.

- It comprises a number of interrelated systems. Each one can be worked on separately, but at some point needs to be integrated back into the entire system.

- It requires significant domain knowledge.

Machine learning at a glance

The physical design process (involving humans, decisions, constraints, and the most potent of all: unpredictability) has parallels with the machine learning systems we are building. The decision boundary of a classifier, data constraints, and the uses of randomness to initialize or introduce diversity in models are just three connections we can make. The deeper question is how far can we take this analogy. If we are trying to build artificial intelligence, the question is, "Are we trying to replicate the process of human intelligence, or simply imitate its consequences, that is, make a reasonable decision?" This of course is ripe for vigorous philosophical discussion and, though interesting, is largely irrelevant to the present discussion. The important point, however, is that much can be learned from observing natural systems, such as the brain, and attempting to mimic their actions.

Real human decision making occurs in a wider context of complex brain action, and in the setting of a design process, the decisions we make are often group decisions. The analogy to an artificial neural net ensemble is irresistible. Like with an ensemble of learning candidates with mostly weak learners, the decisions made, over the lifespan of a project, will end up with a result far greater than any individual contribution. Importantly, an incorrect decision, analogous say to a poor split in a decision tree, is not wasted time since part of the role of weak learners is to rule out incorrect possibilities. In a complex machine learning project, it can be frustrating to realize that much of the work done does not directly lead to a successful result. The initial focus should be on providing convincing arguments that a positive result is possible.

The analogy between machine learning systems and the design process itself is, of course, over simplistic. There are many things in team dynamics that are not represented by a machine learning ensemble. For example, human decision making occurs in the rather illusive context of emotion, intuition, and a lifetime of experience. Also, team dynamics are often shaped by personnel ambition, subtle prejudices, and by relationships between team members. Importantly, managing a team must be integrated into the design process.

A machine learning project of any scale will require collaboration. The space is simply too large for any one person to be fully cognizant of all the different interrelated elements. Even the simple demonstration tasks outlined in this book would not be possible if it were not for the effort of many people developing the theory, writing the base algorithms, and collecting and organizing data.

Successfully orchestrating a major project within time and resource constraints requires significant skill, and these are not necessarily the skills of a software engineer or a data scientist. Obviously, we must define what success, in any given context, means. A theoretical research project either disproving or proving a particular theory with a degree of certainty, or a small degree of uncertainty, is considered a success. Understanding the constraints may give us realistic expectations, in other words, an achievable metric of success.

One of the most common and persistent constraints is that of insufficient, or inaccurate, data. The data collection methodology is such an important aspect, yet in many projects it is overlooked. The data collection process is interactive. It is impossible to interrogate any dynamic system without changing that system. Also, some components of a system are simply easier to observe than others, and therefore, may become inaccurate representations of wider unobserved, or unobservable, components. In many cases, what we know about a complex system is dwarfed by what we do not know. This uncertainty is embedded in the stochastic nature of physical reality, and it is the reason that we must resort to probabilities in any predictive task. Deciding what level of probability is acceptable for a given action, say to treat a potential patient based on the estimated probability of a disease, depends on the consequences of treating the disease or not, and this usually relies on humans, either the doctor or the patient, to make the final decision. There are many issues outside the domain that may influence such a decision.

Human problem solving, although sharing many similarities, is the fundamental difference from machine problem solving. It is dependent on so many things, not least of which is the emotional and physical state, that is, the chemical and electrical bath a nervous system is enveloped in. Human thought is not a deterministic process, and this is actually a good thing because it enables us to solve problems in novel ways. Creative problem solving involves the ability to link disparate ideas or concepts. Often, the inspiration for this comes from an entirely irrelevant event, the proverbial Newton's apple. The ability of the human brain to knit these often random events of every day experience into some sort of coherent, meaningful structure is the illusive ability we aspire to build into our machines.

Summary

There is no doubt that the hardest thing to do in machine learning is to apply it to unique, previously unsolved problems. We have experimented with numerous example models and used some of the most popular algorithms for machine learning. The challenge is now to apply this knowledge to important new problems that you care about. I hope this book has taken you some way as an introduction to the possibilities of machine learning with Python.

Module 3

Advanced Machine Learning with Python

Leverage benefits of machine learning techniques using Python

1
Unsupervised Machine Learning

In this chapter, you will learn how to apply unsupervised learning techniques to identify patterns and structure within datasets.

Unsupervised learning techniques are a valuable set of tools for exploratory analysis. They bring out patterns and structure within datasets, which yield information that may be informative in itself or serve as a guide to further analysis. It's critical to have a solid set of unsupervised learning tools that you can apply to help break up unfamiliar or complex datasets into actionable information.

We'll begin by reviewing **Principal Component Analysis** (**PCA**), a fundamental data manipulation technique with a range of dimensionality reduction applications. Next, we will discuss **k-means clustering**, a widely-used and approachable unsupervised learning technique. Then, we will discuss Kohenen's **Self-Organizing Map** (**SOM**), a method of topological clustering that enables the projection of complex datasets into two dimensions.

Throughout the chapter, we will spend some time discussing how to effectively apply these techniques to make high-dimensional datasets readily accessible. We will use the **UCI Handwritten Digits** dataset to demonstrate technical applications of each algorithm. In the course of discussing and applying each technique, we will review practical applications and methodological questions, particularly regarding how to calibrate and validate each technique as well as which performance measures are valid. To recap, then, we will be covering the following topics in order:

- Principal component analysis
- k-means clustering
- Self-organizing maps

Principal component analysis

In order to work effectively with high-dimensional datasets, it is important to have a set of techniques that can reduce this dimensionality down to manageable levels. The advantages of this dimensionality reduction include the ability to plot multivariate data in two dimensions, capture the majority of a dataset's informational content within a minimal number of features, and, in some contexts, identify collinear model components.

> For those in need of a refresher, collinearity in a machine learning context refers to model features that share an approximately linear relationship. For reasons that will likely be obvious, these features tend to be unhelpful as the related features are unlikely to add information mutually that either one provides independently. Moreover, collinear features may emphasize local minima or other false leads.

Probably the most widely-used dimensionality reduction technique today is PCA. As we'll be applying PCA in multiple contexts throughout this book, it's appropriate for us to review the technique, understand the theory behind it, and write Python code to effectively apply it.

PCA – a primer

PCA is a powerful decomposition technique; it allows one to break down a highly multivariate dataset into a set of orthogonal components. When taken together in sufficient number, these components can explain almost all of the dataset's variance. In essence, these components deliver an abbreviated description of the dataset. PCA has a broad set of applications and its extensive utility makes it well worth our time to cover.

> Note the slightly cautious phrasing here—a given set of components of length less than the number of variables in the original dataset will almost always lose some amount of the information content within the source dataset. This lossiness is typically minimal, given enough components, but in cases where small numbers of principal components are composed from very high-dimensional datasets, there may be substantial lossiness. As such, when performing PCA, it is always appropriate to consider how many components will be necessary to effectively model the dataset in question.

PCA works by successively identifying the axis of greatest variance in a dataset (the principal components). It does this as follows:

1. Identifying the center point of the dataset.
2. Calculating the covariance matrix of the data.
3. Calculating the eigenvectors of the covariance matrix.
4. Orthonormalizing the eigenvectors.
5. Calculating the proportion of variance represented by each eigenvector.

Let's unpack these concepts briefly:
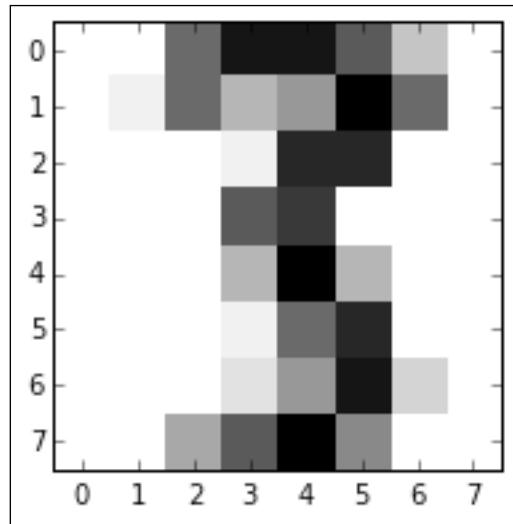
- **Covariance** is effectively variance applied to multiple dimensions; it is the variance between two or more variables. While a single value can capture the variance in one dimension or variable, it is necessary to use a *2 x 2* matrix to capture the covariance between two variables, a *3 x 3* matrix to capture the covariance between three variables, and so on. So the first step in PCA is to calculate this covariance matrix.

- An **Eigenvector** is a vector that is specific to a dataset and linear transformation. Specifically, it is the vector that does not change in direction before and after the transformation is performed. To get a better feeling for how this works, imagine that you're holding a rubber band, straight, between both hands. Let's say you stretch the band out until it is taut between your hands. The eigenvector is the vector that did not change direction between before the stretch and during it; in this case, it's the vector running directly through the center of the band from one hand to the other.

- **Orthogonalization** is the process of finding two vectors that are orthogonal (at right angles) to one another. In an n-dimensional data space, the process of orthogonalization takes a set of vectors and yields a set of orthogonal vectors.

- **Orthonormalization** is an orthogonalization process that also normalizes the product.

- **Eigenvalue** (roughly corresponding to the length of the eigenvector) is used to calculate the proportion of variance represented by each eigenvector. This is done by dividing the eigenvalue for each eigenvector by the sum of eigenvalues for all eigenvectors.

In summary, the covariance matrix is used to calculate Eigenvectors. An orthonormalization process is undertaken that produces orthogonal, normalized vectors from the Eigenvectors. The eigenvector with the greatest eigenvalue is the first principal component with successive components having smaller eigenvalues. In this way, the PCA algorithm has the effect of taking a dataset and transforming it into a new, lower-dimensional coordinate system.

Employing PCA

Now that we've reviewed the PCA algorithm at a high level, we're going to jump straight in and apply PCA to a key Python dataset—the UCI handwritten `digits` dataset, distributed as part of **scikit-learn**.

This dataset is composed of *1,797* instances of handwritten digits gathered from 44 different writers. The input (pressure and location) from these authors' writing is resampled twice across an *8 x 8* grid so as to yield maps of the kind shown in the following image:

These maps can be transformed into feature vectors of length 64, which are then readily usable as analysis input. With an input dataset of 64 features, there is an immediate appeal to using a technique like PCA to reduce the set of variables to a manageable amount. As it currently stands, we cannot effectively explore the dataset with exploratory visualization!

We will begin applying PCA to the handwritten `digits` dataset with the following code:

```
import numpy as np
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
from sklearn.lda import LDA
import matplotlib.cm as cm

digits = load_digits()
data = digits.data

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target
```

This code does several things for us:

1. First, it loads up a set of necessary libraries, including `numpy`, a set of components from scikit-learn, including the `digits` dataset itself, PCA and data scaling functions, and the plotting capability of matplotlib.

2. The code then begins preparing the `digits` dataset. It does several things in order:
    ° First, it loads the dataset before creating helpful variables
    ° The `data` variable is created for subsequent use, and the number of distinct `digits` in the `target` vector (*0* through to *9*, so `n_digits = 10`) is saved as a variable that we can easily access for subsequent analysis
    ° The `target` vector is also saved as labels for later use
    ° All of this variable creation is intended to simplify subsequent analysis

3. With the dataset ready, we can initialize our PCA algorithm and apply it to the dataset:

```
pca = PCA(n_components=10)
data_r = pca.fit(data).transform(data)

print('explained variance ratio (first two components): %s' %
str(pca.explained_variance_ratio_))
print('sum of explained variance (first two components): %s' %
str(sum(pca.explained_variance_ratio_)))
```

4. This code outputs the variance explained by each of the first ten principal components ordered by explanatory power.
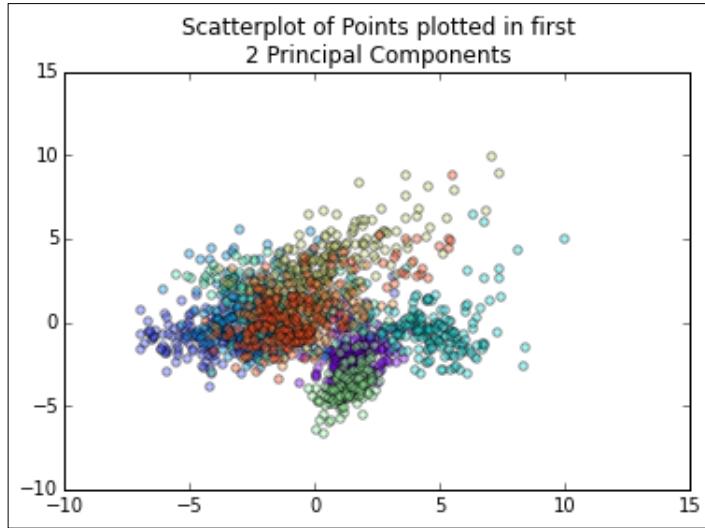
In the case of this set of 10 principal components, they collectively explain *0.589* of the overall dataset variance. This isn't actually too bad, considering that it's a reduction from *64* variables to 10 components. It does, however, illustrate the potential lossiness of PCA. The key question, though, is whether this reduced set of components makes subsequent analysis or classification easier to achieve; that is, whether many of the remaining components contained variance that disrupts classification attempts.

Having created a `data_r` object containing the output of `pca` performed over the `digits` dataset, let's visualize the output. To do so, we'll first create a vector of `colors` for class coloration. We then simply create a scatterplot with colorized classes:

```
X = np.arange(10)
ys = [i+x+(i*x)**2 for i in range(10)]

plt.figure()
colors = cm.rainbow(np.linspace(0, 1, len(ys)))
for c, i target_name in zip(colors, [1,2,3,4,5,6,7,8,9,10], labels):
    plt.scatter(data_r[labels == I, 0], data_r[labels == I, 1],
    c=c, alpha = 0.4)
    plt.legend()
    plt.title('Scatterplot of Points plotted in first \n'
    '10 Principal Components')
    plt.show()
```

The resulting scatterplot looks as follows:

This plot shows us that, while there is some separation between classes in the first two principal components, it may be tricky to classify highly accurately with this dataset. However, classes do appear to be clustered and we may be able to get reasonably good results by employing a clustering analysis. In this way, PCA has given us some insight into how the dataset is structured and has informed our subsequent analysis.

At this point, let's take this insight and move on to examine clustering by the application of the k-means clustering algorithm.

Introducing k-means clustering

In the previous section, you learned that unsupervised machine learning algorithms are used to extract key structural or information content from large, possibly complex datasets. These algorithms do so with little or no manual input and function without the need for training data (sets of labeled explanatory and response variables needed to train an algorithm in order to recognize the desired classification boundaries). This means that unsupervised algorithms are effective tools to generate information about the structure and content of new or unfamiliar datasets. They allow the analyst to build a strong understanding in a fraction of the time.

Clustering – a primer

Clustering is probably the archetypal unsupervised learning technique for several reasons.

A lot of development time has been sunk into optimizing clustering algorithms, with efficient implementations available in most data science languages including Python.

Clustering algorithms tend to be very fast, with smoothed implementations running in polynomial time. This makes it uncomplicated to run multiple clustering configurations, even over large datasets. Scalable clustering implementations also exist that parallelize the algorithm to run over **TB-scale** datasets.

Clustering algorithms are frequently easily understood and their operation is thus easy to explain if necessary.

The most popular clustering algorithm is k-means; this algorithm forms k-many clusters by first randomly initiating the clusters as k-many points in the data space. Each of these points is the mean of a cluster. An iterative process then occurs, running as follows:

- Each point is assigned to a cluster based on the least (within cluster) sum of squares, which is intuitively the nearest mean.
- The center (centroid) of each cluster becomes the new mean. This causes each of the means to shift.

Over enough iterations, the centroids move into positions that minimize a performance metric (the performance metric most commonly used is the "within cluster least sum of squares" measure). Once this measure is minimized, observations are no longer reassigned during iteration; at this point the algorithm has converged on a solution.

Kick-starting clustering analysis

Now that we've reviewed the clustering algorithm, let's run through the code and see what clustering can do for us:

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

np.random.seed()



digits = load_digits()
```

```
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

sample_size = 300

print("n_digits: %d, \t n_samples %d, \t n_features %d"
   % (n_digits, n_samples, n_features))


print(79 * '_')
print('% 9s' % 'init''         time   inertia   homo   compl   v-meas
ARI     AMI  silhouette')

def bench_k_means(estimator, name, data):
   t0 = time()
   estimator.fit(data)
   print('% 9s %.2fs %i %.3f %.3f %.3f %.3f %.3f %.3f'
      % (name, (time() - t0), estimator.inertia_,
         metrics.homogeneity_score(labels, estimator.labels_),
         metrics.completeness_score(labels, estimator.labels_),
         metrics.v_measure_score(labels, estimator.labels_),
         metrics.adjusted_rand_score(labels, estimator.labels_),
         metrics.silhouette_score(data, estimator.labels_,
            metric='euclidean',
            sample_size=sample_size)))
```

> One critical difference between this code and the PCA code we saw previously is that this code begins by applying a scale function to the `digits` dataset. This function scales values in the dataset between *0* and *1*. It's critically important to scale data wherever needed, either on a log scale or bound scale, so as to prevent the magnitude of different feature values to have disproportionately powerful effects on the dataset. The key to determining whether the data needs scaling at all (and what kind of scaling is needed, within which range, and so on) is very much tied to the shape and nature of the data. If the distribution of the data shows outliers or variation within a large range, it may be appropriate to apply log-scaling. Whether this is done manually through visualization and exploratory analysis techniques or through the use of summary statistics, decisions around scaling are tied to the data under inspection and the analysis techniques to be used. A further discussion of scaling decisions and considerations may be found in *Chapter 7, Feature Engineering Part II*.

Helpfully, scikit-learn uses the k-means++ algorithm by default, which improves over the original k-means algorithm in terms of both running time and success rate in avoiding poor clusterings.

The algorithm achieves this by running an initialization procedure to find cluster centroids that approximate minimal variance within classes.

You may have spotted from the preceding code that we're using a set of performance estimators to track how well our k-means application is performing. It isn't practical to measure the performance of a clustering algorithm based on a single correctness percentage or using the same performance measures that are commonly used with other algorithms. The definition of success for clustering algorithms is that they provide an interpretation of how input data is grouped that trades off between several factors, including class separation, in-group similarity, and cross-group difference.

The **homogeneity score** is a simple, zero-to-one-bounded measure of the degree to which clusters contain only assignments of a given class. A score of one indicates that all clusters contain measurements from a single class. This measure is complimented by the **completeness score**, which is a similarly bounded measure of the extent to which all members of a given class are assigned to the same cluster. As such, a completeness score and homogeneity score of one indicates a perfect clustering solution.

The **validity measure (v-measure)** is a harmonic mean of the homogeneity and completeness scores, which is exactly analogous to the F-measure for binary classification. In essence, it provides a single, 0-1-scaled value to monitor both homogeneity and completeness.

The **Adjusted Rand Index** (**ARI**) is a similarity measure that tracks the consensus between sets of assignments. As applied to clustering, it measures the consensus between the true, pre-existing observation labels and the labels predicted as an output of the clustering algorithm. The Rand index measures labeling similarity on a *0-1* bound scale, with one equaling perfect prediction labels.

The main challenge with all of the preceding performance measures as well as other similar measures (for example, Akaike's mutual information criterion) is that they require an understanding of the ground truth, that is, they require some or all of the data under inspection to be labeled. If labels do not exist and cannot be generated, these measures won't work. In practice, this is a pretty substantial drawback as very few datasets come prelabeled and the creation of labels can be time-consuming.

One option to measure the performance of a k-means clustering solution without labeled data is the **Silhouette Coefficient**. This is a measure of how well-defined the clusters within a model are. The Silhouette Coefficient for a given dataset is the mean of the coefficient for each sample, where this coefficient is calculated as follows:

$$s = \frac{b - a}{\max(a, b)}$$

The definitions of each term are as follows:

- *a*: The mean distance between a sample and all other points in the same cluster
- *b*: The mean distance between a sample and all other points in the next nearest cluster

This score is bounded between *-1* and *1*, with *-1* indicating incorrect clustering, *1* indicating very dense clustering, and scores around *0* indicating overlapping clusters. This tends to fit our expectations of how a good clustering solution is composed.

In the case of the `digits` dataset, we can employ all of the performance measures described here. As such, we'll complete the preceding example by initializing our `bench_k_means` function over the `digits` dataset:

```
bench_k_means(KMeans(init='k-means++', n_clusters=n_digits, n_
init=10), name="k-means++", data=data)
print(79 * '_')
```

This yields the following output (note that the random seed means your results will vary from mine!):

n_digits: 10,		n_samples 1797,			n_features 64			
init	time	inertia	homo	compl	v-meas	ARI	AMI	silhouette
k-means++	0.25s	69517	0.596	0.643	0.619	0.465	0.592	0.123

Lets take a look at these results in more detail.

The Silhouette score at `0.123` is fairly low, but not surprisingly so, given that the handwritten digits data is inherently noisy and does tend to overlap. However, some of the other scores are not that impressive. The V-measure at `0.619` is reasonable, but in this case is held back by a poor homogeneity measure, suggesting that the cluster centroids did not resolve perfectly. Moreover, the ARI at `0.465` is not great.

> Let's put this in context. The worst case classification attempt, random assignment, would give at best 10% classification accuracy. All of our performance measures would be accordingly very low. While we're definitely doing a lot better than that, we're still trailing far behind the best computational classification attempts. As we'll see in *Chapter 4*, *Convolutional Neural Networks*, convolutional nets achieve results with extremely low classification errors on handwritten digit datasets. We're unlikely to achieve this level of accuracy with traditional k-means clustering!

All in all, it's reasonable to think that we could do better.

To give this another try, we'll apply an additional stage of processing. To learn how to do this, we'll apply PCA—the technique we previously walked through—to reduce the dimensionality of our input dataset. The code to achieve this is very simple, as follows:

```
pca = PCA(n_components=n_digits).fit(data)
bench_k_means(KMeans(init=pca.components_, n_clusters=10),
name="PCA-based",
data=data)
```
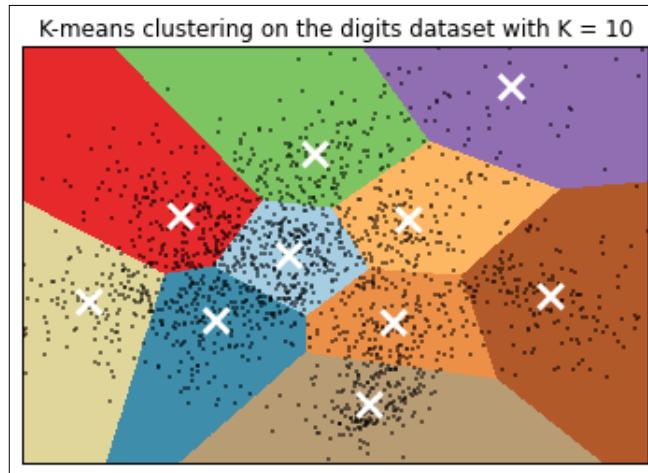
This code simply applies PCA to the digits dataset, yielding as many principal components as there are classes (in this case, digits). It can be sensible to review the output of PCA before proceeding as the presence of any small principal components may suggest a dataset that contains collinearity or otherwise merits further inspection.

This instance of clustering shows noticeable improvement:

n_digits: 10,		n_samples 1797,			n_features 64		
init	time	inertia	homo	compl	v-meas	ARI	silhouette
PCA-based	0.02s	71820	0.673	0.715	0.693	0.567	0.121

The V-measure and ARI have increased by approximately *0.08* points, with the V-measure reading a fairly respectable 0.693. The Silhouette Coefficient did not change significantly. Given the complexity and interclass overlap within the digits dataset, these are good results, particularly stemming from such a simple code addition!

Inspection of the `digits` dataset with clusters superimposed shows that some meaningful clusters appear to have been formed. It is also apparent from the following plot that actually detecting the character from the input feature vectors may be a challenging task:

K-means clustering on the digits dataset with K = 10

Tuning your clustering configurations

The previous examples described how to apply k-means, walked through relevant code, showed how to plot the results of a clustering analysis, and identified appropriate performance metrics. However, when applying k-means to real-world datasets, there are some extra precautions that need to be taken, which we will discuss.

Another critical practical point is how to select an appropriate value for k. Initializing k-means clustering with a specific k value may not be harmful, but in many cases it is not clear initially how many clusters you might find or what values of k may be helpful.

We can rerun the preceding code for multiple values of k in a batch and look at the performance metrics, but this won't tell us which instance of k is most effectively capturing structure within the data. The risk is that as k increases, the Silhouette Coefficient or unexplained variance may decrease dramatically, without meaningful clusters being formed. The extreme case of this would be if $k = o$, where o is the number of observations in the sample; every point would have its own cluster, the Silhouette Coefficient would be low, but the results wouldn't be meaningful. There are, however, many less extreme cases in which overfitting may occur due to an overly high k value.

To mitigate this risk, it's advisable to use supporting techniques to motivate a selection of *k*. One useful technique in this context is the **elbow method**. The elbow method is a very simple technique; for each instance of *k*, plot the percentage of explained variance against *k*. This typically leads to a plot that frequently looks like a bent arm.

For the PCA-reduced dataset, this code looks like the following snippet:
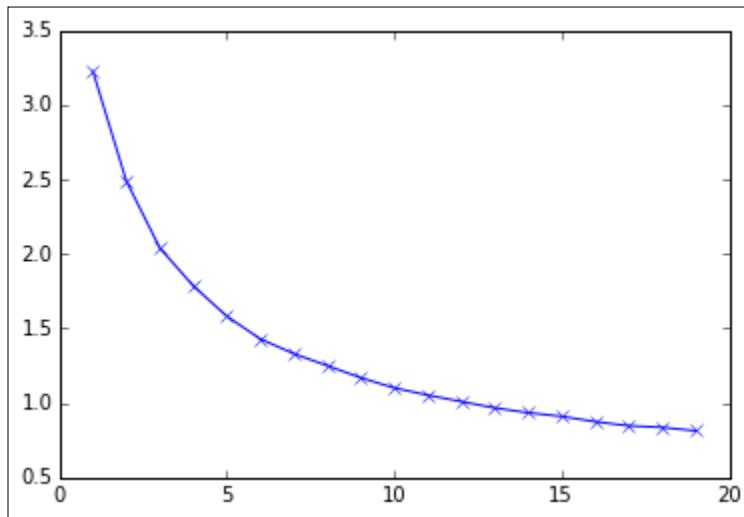
```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

K = range(1,20)
explainedvariance= []
for k in K:
    reduced_data = PCA(n_components=2).fit_transform(data)
    kmeans = KMeans(init = 'k-means++', n_clusters = k, n_init = k)
    kmeans.fit(reduced_data)
    explainedvariance.append(sum(np.min(cdist(reduced_data,
    kmeans.cluster_centers_, 'euclidean'), axis =
    1))/data.shape[0])
    plt.plot(K, meandistortions, 'bx-')
    plt.show()
```

This application of the elbow method takes the `PCA` reduction from the previous code sample and applies a test of the explained variance (specifically, a test of the variance within clusters). The result is output as a measure of unexplained variance for each value of *k* in the range specified. In this case, as we're using the `digits` dataset (which we know to have ten classes), the `range` specified was 1 to 20:
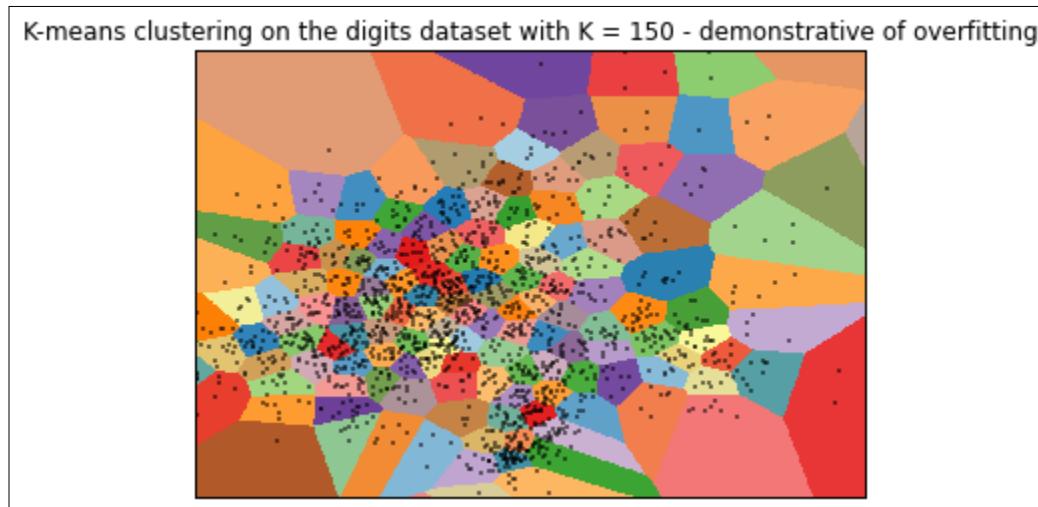
The elbow method involves selecting the value of *k* that maximizes explained variance while minimizing κ; that is, the value of *k* at the crook of the elbow. The technical sense underlying this is that a minimal gain in explained variance at greater values of *k* is offset by the increasing risk of overfitting.

Elbow plots may be more or less pronounced and the elbow may not always be clearly identifiable. This example shows a more gradual progression than may be observable in other cases with other datasets. It's worth noting that, while we know the number of classes within the dataset to be ten, the elbow method starts to show diminishing returns on *k* increases almost immediately and the elbow is located at around five classes. This has a lot to do with the substantial overlap between classes, which we saw in previous plots. While there are ten classes, it becomes increasingly difficult to clearly identify more than five or so.

With this in mind, it's worth noting that the elbow method is intended for use as a heuristic rather than as some kind of objective principle. The use of PCA as a preprocess to improve clustering performance also tends to smooth the graph, delivering a more gradual curve than otherwise.

In addition to making use of the elbow method, it can be valuable to look at the clusters themselves, as we did earlier in the chapter, using PCA to reduce the dimensionality of the data. By plotting the dataset and projecting cluster assignation onto the data, it is sometimes very obvious when a k-means implementation has fitted to a local minima or has overfit the data. The following plot demonstrates extreme overfitting of our previous k-means clustering algorithm to the `digits` dataset, artificially prompted by using **K = 150**. In this example, some clusters contain a single observation; there's really no way that this output would generalize to other samples well:

K-means clustering on the digits dataset with K = 150 - demonstrative of overfitting

Plotting the elbow function or cluster assignments is quick to achieve and straightforward to interpret. However, we've spoken of these techniques in terms of being heuristics. If a dataset contains a deterministic number of classes, we may not be sure that a heuristic method will deliver generalizable results.

Another drawback is that visual plot checking is a very manual technique, which makes it poorly-suited for production environments or automation. In such circumstances, it's ideal to find a code-based, automatable method. One solid option in this case is **v-fold cross-validation**, a widely-used validation technique.

Cross-validation is simple to undertake. To make it work, one splits the dataset into *v* parts. One of the parts is set aside individually as a test set. The model is trained against the training data, which is all parts except the test set. Let's try this now, again using the `digits` dataset:

```
import numpy as np
from sklearn import cross_validation
from sklearn.cluster import KMeans
```

```
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale


digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target



kmeans = KMeans(init='k-means++', n_clusters=n_digits, n_init=n_
digits)
cv = cross_validation.ShuffleSplit(n_samples, n_iter = 10, test_size =
0.4, random_state = 0)
scores = cross_validation.cross_val_score(kmeans, data, labels, cv =
cv, scoring = 'adjusted_rand_score')
print(scores)
print(sum(scores)/cv.n_iter)
```

This code performs some now familiar data loading and preparation and initializes
the k-means clustering algorithm. It then defines `cv`, the cross-validation parameters.
This includes specification of the number of iterations, `n_iter`, and the amount
of data that should be used in each fold. In this case, we're using 60% of the data
samples as training data and 40% as test data.

We then apply the k-means model and `cv` parameters that we've specified within the
cross-validation scoring function and print the results as `scores`. Let's take a look at
these scores now:

```
[ 0.39276606  0.49571292  0.43933243  0.53573558  0.42459285
  0.55686854  0.4573401   0.49876358  0.50281585  0.4689295 ]


0.4772857426
```

This output gives us, in order, the adjusted Rand score for cross-validated,
k-means++ clustering performed across each of the `10` folds in order. We can see
that results do fluctuate between around `0.4` and `0.55`; the earlier ARI score for
k-means++ without PCA fell within this range (at `0.465`). What we've created, then,
is code that we can incorporate into our analysis in order to check the quality of our
clustering automatically on an ongoing basis.

As noted earlier in this chapter, your choice of success measure is contingent on what information you already have. In most cases, you won't have access to ground truth labels from a dataset and will be obliged to use a measure such as the Silhouette Coefficient that we discussed previously.

> Sometimes, even using both cross-validation and visualizations won't provide a conclusive result. Especially with unfamiliar datasets, it's not unheard of to run into issues where some noise or secondary signal resolves better at a different k value than the signal you're attempting to analyze.
>
> As with every other algorithm discussed in this book, it is imperative to understand the dataset one wishes to work with. Without this insight, it's entirely possible for even a technically correct and rigorous analysis to deliver inappropriate conclusions. *Chapter 6, Text Feature Engineering* will discuss principles and techniques for the inspection and preparation of unfamiliar datasets more thoroughly.

Self-organizing maps

A SOM is a technique to generate topological representations of data in reduced dimensions. It is one of a number of techniques with such applications, with a better-known alternative being PCA. However, SOMs present unique opportunities, both as dimensionality reduction techniques and as a visualization format.

SOM – a primer

The SOM algorithm involves iteration over many simple operations. When applied at a smaller scale, it behaves similarly to k-means clustering (as we'll see shortly). At a larger scale, SOMs reveal the topology of complex datasets in a powerful way.

An SOM is made up of a grid (commonly rectangular or hexagonal) of nodes, where each node contains a weight vector that is of the same dimensionality as the input dataset. The nodes may be initialized randomly, but an initialization that roughly approximates the distribution of the dataset will tend to train faster.

The algorithm iterates as observations are presented as input. Iteration takes the following form:

- Identifying the winning node in the current configuration—the **Best Matching Unit** (**BMU**). The BMU is identified by measuring the Euclidean distance in the data space of all the weight vectors.

- The BMU is adjusted (moved) towards the input vector.

- Neighboring nodes are also adjusted, usually by lesser amounts, with the magnitude of neighboring movement being dictated by a neighborhood function. (Neighborhood functions vary. In this chapter, we'll use a Gaussian neighborhood function.)

This process repeats over potentially many iterations, using sampling if appropriate, until the network converges (reaching a position where presenting a new input does not provide an opportunity to minimize loss).
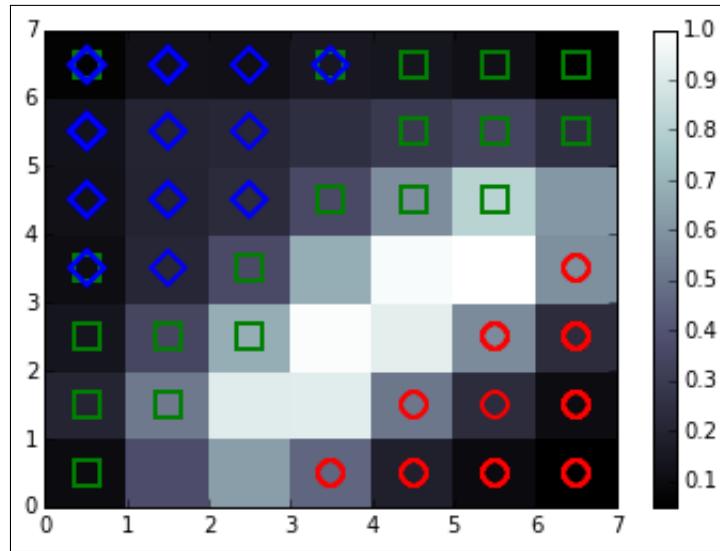
A node in an SOM is not unlike that of a neural network. It typically possesses a weight vector of length equal to the dimensionality of the input dataset. This means that the topology of the input dataset can be preserved and visualized through a lower-dimensional mapping.

The code for this SOM class implementation is available in the book repository in the `som.py` script. For now, let's start working with the SOM algorithm in a familiar context.

Employing SOM

As discussed previously, the SOM algorithm is iterative, being based around Euclidean distance comparisons of vectors.

This mapping tends to form a fairly readable 2D grid. In the case of the commonly-used Iris tutorial dataset, an SOM will map it out pretty cleanly:

In this diagram, the classes have been separated and also ordered spatially. The background coloring in this case is a clustering density measure. There is some minimal overlap between the blue and green classes, where the SOM performed an imperfect separation. On the Iris dataset, an SOM will tend to approach a converged solution on the order of 100 iterations, with little visible improvement after 1,000. For more complex datasets containing less clearly divisible cases, this process can take tens of thousands of iterations.

Awkwardly, there aren't implementations of the SOM algorithm within pre-existing Python packages like scikit-learn. This makes it necessary for us to use our own implementation.

The SOM code we'll be working with for this purpose is located in the associated GitHub repository. For now, let's take a look at the relevant script and get an understanding of how the code works:

```
import numpy as np
from sklearn.datasets import load_digits
from som import Som
```

```
from pylab import plot,axis,show,pcolor,colorbar,bone


digits = load_digits()
data = digits.data
labels = digits.target
```

At this point, we've loaded the `digits` dataset and identified `labels` as a separate set of data. Doing this will enable us to observe how the SOM algorithm separates classes when assigning them to `map`:

```
som = Som(16,16,64,sigma=1.0,learning_rate=0.5)
som.random_weights_init(data)
print("Initiating SOM.")
som.train_random(data,10000)
print("\n. SOM Processing Complete")

bone()
pcolor(som.distance_map().T)
colorbar()
```

At this point, we have utilized a `Som` class that is provided in a separate file, `Som.py`, in the repository. This class contains the methods required to deliver the SOM algorithm we discussed earlier in the chapter. As arguments to this function, we provide the dimensions of the map (After trialing a range of options, we'll start out with 16 x 16 in this case—this grid size gave the feature map enough space to spread out while retaining some overlap between groups.) and the dimensionality of the input data. (This argument determines the length of the weight vector within the SOM's nodes.) We also provide values for sigma and learning rate.
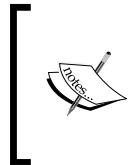
Sigma, in this case, defines the spread of the neighborhood function. As noted previously, we're using a Gaussian neighborhood function. The appropriate value for sigma varies by grid size. For an *8 x 8* grid, we would typically want to use a value of *1.0* for Sigma, while in this case we're using *1.3* for a *16 x 16* grid. It is fairly obvious when one's value for sigma is off; if the value is too small, values tend to cluster near the center of the grid. If the values are too large, the grid typically ends up with several large, empty spaces towards the center.

The *learning rate* self-explanatorily defines the initial learning rate for the SOM. As the map continues to iterate, the learning rate adjusts according to the following function:

$$learning\,rate(t) = learning\,rate \big/ \big(1 + t \big/ (0.5 * t)\big)$$

Here, *t* is the iteration index.

We follow up by first initializing our SOM with random weights.

> As with k-means clustering, this initialization method is slower than initializing based on an approximation of the data distribution. A preprocessing step similar to that employed by the k-means++ algorithm would accelerate the SOM's runtime. Our SOM runs sufficiently quickly over the `digits` dataset to make this optimization unnecessary for now.

Next, we set up label and color assignations for each class, so that we can distinguish classes on the plotted SOM. Following this, we iterate through each data point.
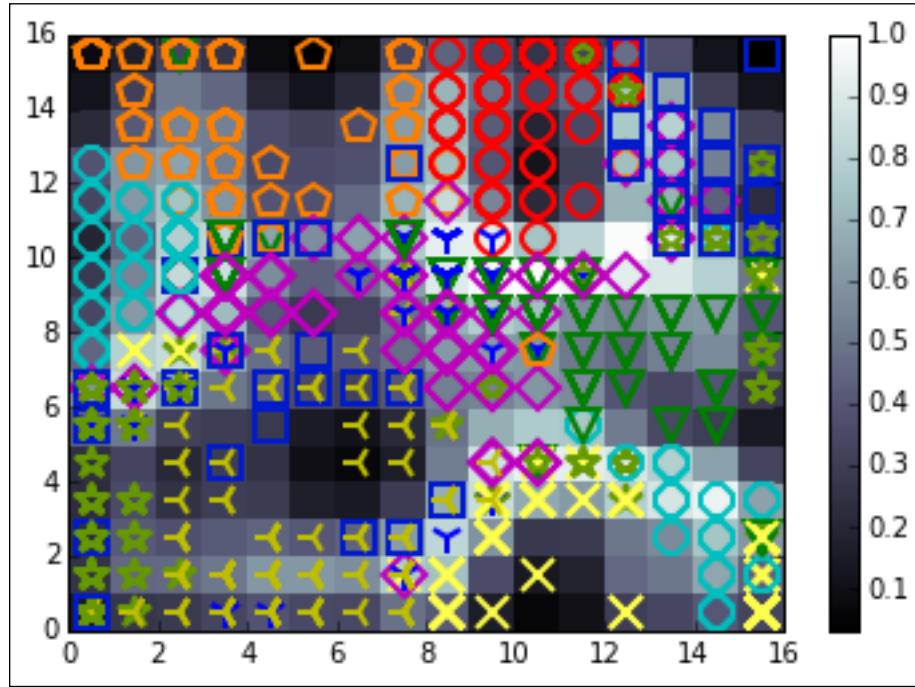
On each iteration, we plot a class-specific marker for the BMU as calculated by our SOM algorithm.

When the SOM finishes iteration, we add a **U-Matrix** (a colorized matrix of relative observation density) as a monochrome-scaled plot layer:

```
labels[labels == '0'] = 0
labels[labels == '1'] = 1
labels[labels == '2'] = 2
labels[labels == '3'] = 3
labels[labels == '4'] = 4
labels[labels == '5'] = 5
labels[labels == '6'] = 6
labels[labels == '7'] = 7
labels[labels == '8'] = 8
labels[labels == '9'] = 9

markers = ['o', 'v', '1', '3', '8', 's', 'p', 'x', 'D', '*']
colors = ["r", "g", "b", "y", "c", (0,0.1,0.8), (1,0.5,0), (1,1,0.3),
"m", (0.4,0.6,0)]
for cnt,xx in enumerate(data):
   w = som.winner(xx)
   plot(w[0]+.5,w[1]+.5,markers[labels[cnt]],
   markerfacecolor='None', markeredgecolor=colors[labels[cnt]],
   markersize=12, markeredgewidth=2)
   axis([0,som.weights.shape[0],0,som.weights.shape[1]])
   show()
```

This code generates a plot similar to the following:

This code delivers a 16 x 16 node SOM plot. As we can see, the map has done a reasonably good job of separating each cluster into topologically distinct areas of the map. Certain classes (particularly the digits five in cyan circles and nine in green stars) have been located over multiple parts of the SOM space. For the most part, though, each class occupies a distinct region and it's fair to say that the SOM has been reasonably effective. The U-Matrix shows that regions with a high density of points are co-habited by data from multiple classes. This isn't really a surprise as we saw similar results with k-means and PCA plotting.

Further reading

Victor Powell and Lewis Lehe provide a fantastic interactive, visual explanation of PCA at `http://setosa.io/ev/principal-component-analysis/`, this is ideal for readers who are new to the core concepts of PCA or who are not quite getting it.

For a lengthier and more mathematically-involved treatment of PCA, touching on underlying matrix transformations, Jonathon Shlens from Google research provides a clear and thorough explanation at `http://arxiv.org/abs/1404.1100`.

For a thorough worked example that translates Jonathon's description into clear Python code, consider Sebastian Raschka's demonstration using the Iris dataset at `http://sebastianraschka.com/Articles/2015_pca_in_3_steps.html`.

Finally, consider the sklearn documentation for more details on arguments to the PCA class at `http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html`.

For a lively and expert treatment of k-means, including detailed investigations of the conditions that cause it to fail, and potential alternatives in such cases, consider David Robinson's fantastic blog, variance explained at `http://varianceexplained.org/r/kmeans-free-lunch/`.

A specific discussion of the Elbow method is provided by Rick Gove at `https://bl.ocks.org/rpgove/0060ff3b656618e9136b`.

Finally, consider sklearn's documentation for another view on unsupervised learning algorithms, including k-means at `http://scikit-learn.org/stable/tutorial/statistical_inference/unsupervised_learning.html`.

Much of the existing material on Kohonen's SOM is either rather old, very high-level, or formally expressed. A decent alternative to the description in this book is provided by John Bullinaria at `http://www.cs.bham.ac.uk/~jxb/NN/l16.pdf`.

For readers interested in a deeper understanding of the underlying mathematics, I'd recommend reading the work of Tuevo Kohonen directly. The 2012 edition of self-organising maps is a great place to start.

The concept of multicollinearity, referenced in the chapter, is given a clear explanation for the unfamiliar at `https://onlinecourses.science.psu.edu/stat501/node/344`.

Summary

In this chapter, we've reviewed three techniques with a broad range of applications for preprocessing and dimensionality reduction. In doing so, you learned a lot about an unfamiliar dataset.

We started out by applying PCA, a widely-utilized dimensionality reduction technique, to help us understand and visualize a high-dimensional dataset. We then followed up by clustering the data using k-means clustering, identifying means of improving and measuring our k-means analysis through performance metrics, the elbow method, and cross-validation. We found that k-means on the `digits` dataset, taken as is, didn't deliver exceptional results. This was due to class overlap that we spotted through PCA. We overcame this weakness by applying PCA as a preprocess to improve our subsequent clustering results.

Finally, we developed an SOM algorithm that delivered a cleaner separation of the `digit` classes than PCA.

Having learned some key basics around unsupervised learning techniques and analytical methodology, let's dive into the use of some more powerful unsupervised learning algorithms.

2
Deep Belief Networks

In the preceding chapter, we looked at some widely-used dimensionality reduction techniques, which enable a data scientist to get greater insight into the nature of datasets.

The next few chapters will focus on some more sophisticated techniques, drawing from the area of deep learning. This chapter is dedicated to building an understanding of how to apply the **Restricted Boltzmann Machine** (**RBM**) and manage the deep learning architecture one can create by chaining RBMs—the **deep belief network** (**DBN**). DBNs are trainable to effectively solve complex problems in text, image, and sound recognition. They are used by leading companies for object recognition, intelligent image search, and robotic spatial recognition.

The first thing that we're going to do is get a solid grounding in the algorithm underlying DBN; unlike clustering or PCA, this code isn't widely-known by data scientists and we're going to review it in some depth to build a strong working knowledge. Once we've worked through the theory, we'll build upon it by stepping through code that brings the theory into focus and allows us to apply the technique to real-world data. The diagnosis of these techniques is not trivial and needs to be rigorous, so we'll emphasize the thought processes and diagnostic techniques that enable us to effectively watch and control the success of your implementation.

By the end of this chapter, you'll understand how the RBM and DBN algorithms work, know how to use them, and feel confident in your ability to improve the quality of the results you get out of them. To summarize, the contents of this chapter are as follows:

- Neural networks – a primer
- Restricted Boltzmann Machines
- Deep belief networks

Neural networks – a primer

The RBM is a form of recurrent neural network. In order to understand how the RBM works, it is necessary to have a more general understanding of neural networks. Readers with an understanding of artificial neural network (hereafter neural network, for the sake of simplicity) algorithms will find familiar elements in the following description.

There are many accounts that cover neural networks in great theoretical detail; we won't go into great detail retreading this ground. For the purposes of this chapter, we will first describe the components of a neural network, common architectures, and prevalent learning processes.

The composition of a neural network

For unfamiliar readers, neural networks are a class of mathematical models that train to produce and optimize a definition for a function (or distribution) over a set of input features. The specific objective of a given neural network application can be defined by the operator using a performance measure (typically a cost function); in this way, neural networks may be used to classify, predict, or transform their inputs.

The use of the word neural in neural networks is the product of a long tradition of drawing from heavy-handed biological metaphors to inspire machine learning research. Hence, artificial neural networks algorithms originally drew (and frequently still draw) from biological neuronal structures.

A neural network is composed of the following elements:

- A learning process: A neural network learns by adjusting parameters within the weight function of its nodes. This occurs by feeding the output of a performance measure (as described previously, in supervised learning contexts this is frequently a cost function, some measure of inaccuracy relative to the target output of the network) into the learning function of the network. This learning function outputs the required weight adjustments (Technically, it typically calculates the partial derivatives—terms required by gradient descent.) to minimize the cost function.

- A set of neurons or weights: Each contains a weight function (the activation function) that manipulates input data. The activation function may vary substantially between networks (with one well-known example being the hyperbolic tangent). The key requirement is that the weights must be adaptive, that is,, adjustable based on updates from the learning process. In order to model non-parametrically (that is, to model effectively without defining details of the probability distribution), it is necessary to use both visible and hidden units. Hidden units are never observed.

- Connectivity functions: They control which nodes can relay data to which other nodes. Nodes may be able to freely relay input to one another in an unrestricted or restricted fashion, or they may be more structured in layers through which input data must flow in a directed fashion. There is a broad range of interconnection patterns, with different patterns producing very different network properties and possibilities.
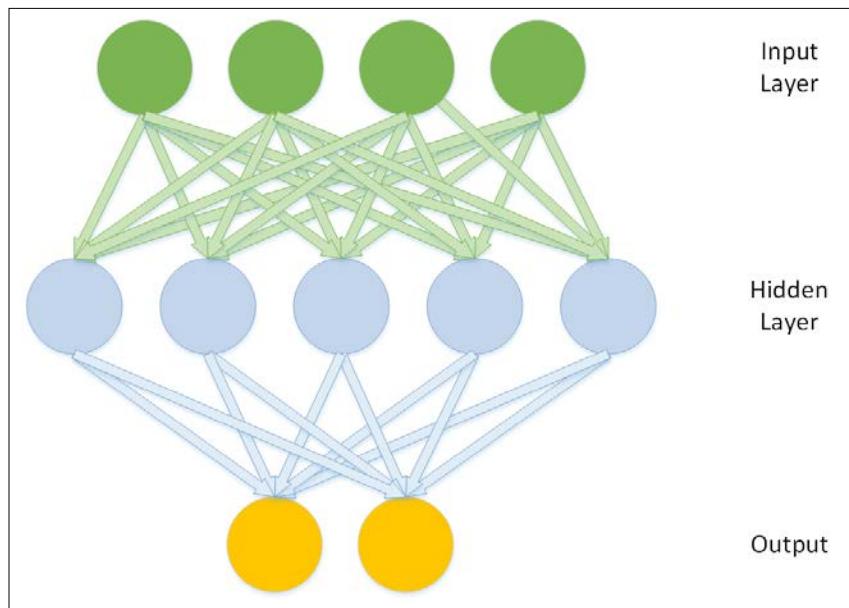
Utilizing this set of elements enables us to build a broad range of neural networks, ranging from the familiar directed acyclic graph (with perhaps the best-known example being the **Multi-Layer Perceptron** (**MLP**)) to creative alternatives. The **Self-Organizing Map** (**SOM**) that we employed in the preceding chapter was a type of neural network, with a unique learning process. The algorithm that we'll examine later in this chapter, that of the RBM, is another neural network algorithm with some unique properties.

Network topologies
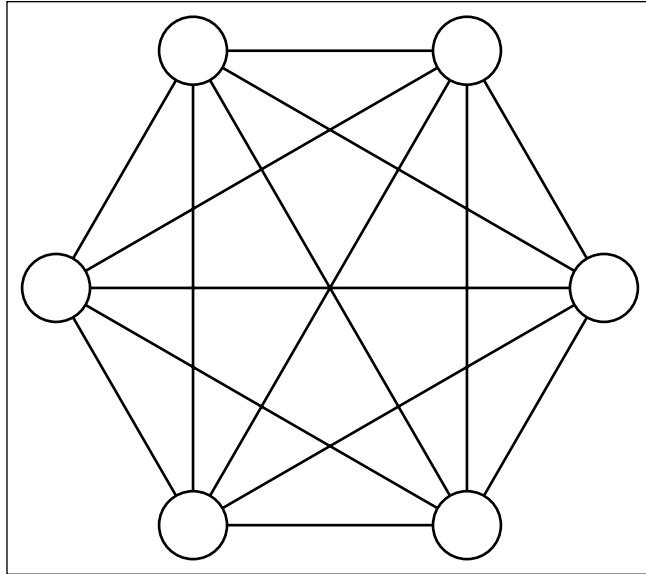
There are many variations on how the neurons in a neural network are connected, with structural decisions being an important factor in determining the network's learning capabilities. Common topologies in unsupervised learning tend to differ from those common to supervised learning. One common and now familiar unsupervised learning topology is that of the SOM that we discussed in the last chapter.

The SOM, as we saw, directly projects individual input cases onto a weight vector contained by each node. It then proceeds to reorder these nodes until an appropriate mapping of the dataset is converged on. The actual structure of the SOM was a variant based on the details of training, specific outcome of a given instance of training, and design decisions taken in structuring the network, but square or hexagonal grid structures are becoming increasingly common.

A very common topology type in supervised learning is that of a three-layer, feedforward network, with the classical case being the MLP. In this network topology model, the neurons in the network are split into layers, with each layer communicating to the layer "beyond" it. The first layer contains inputs that are fed to a hidden layer. The hidden layer develops a representation of the data using weight activations (with the right activation function, for example, sigmoid or gauss, an MLP can act as a universal function approximator) and activation values are communicated to the output layer. The output layer typically delivers network results. This topology, therefore, looks as follows:

Other network topologies deliver different capabilities. The topology of a Boltzmann Machine, for instance, differs from those described previously. The Boltzmann machine contains hidden and visible neurons, like those of a three-layer network, but all of these neurons are connected to one another in a directed, cyclic graph:

This topology makes Boltzmann machines stochastic—probabilistic rather than deterministic—and able to develop in one of several ways given a sufficiently complex problem. The Boltzmann machine is also generative, which means that it is able to fully (probabilistically) model all of the input variables, rather than using the observed variables to specifically model the target variables.

Which network topology is appropriate depends to a large extent on your specific challenge and the desired output. Each tends to be strong in certain areas. Furthermore, each of the topologies described here will be accompanied by a learning process that enables the network to iteratively converge on an (ideally optimal) solution.

There are a broad range of learning processes, with specific processes and topologies being more or less compatible with one another. The purpose of a learning process is to enable the network to adjust its weights, iteratively, in such a way as to create an increasingly accurate representation of the input data.

As with network topologies, there are a great many learning processes to consider. Some familiarity is assumed and a great many excellent resources on learning processes exist (some good examples are given at the end of this chapter). This section will focus on delivering a common characterization of learning processes, while later in the chapter, we'll look in greater detail at a specific example.

As noted, the objective of learning in a neural network is to iteratively improve the distribution of weights across the model so that it approximates the function underlying input data with increasing accuracy. This process requires a performance measure. This may be a classification error measure, as is commonly used in supervised, classification contexts (that is, with the backpropagation learning algorithm in MLP networks). In stochastic networks, it may be a probability maximization term (such as energy in energy-based networks).

In either case, once there is a measure to increase probability, the network is effectively attempting to reduce that measure using an optimization method. In many cases, the optimization of the network is achieved using **gradient descent**. As far as the gradient descent algorithm method is concerned, the size of your performance measure value on a given training iteration is analogous to the slope of your gradient. Minimizing the performance measure is therefore a question of descending that gradient to the point at which the error measure is at its lowest for that set of weights.

The size of the network's updates for the next iteration (the learning rate of your algorithm) may be influenced by the magnitude of your performance measure, or it may be hard-coded.

The weight updates by which your network adjusts may be derived from the error surface itself; if so, your network will typically have a means of calculating the gradient, that is, deriving the values to which updates need to adjust the parameters on your network's activated weight functions so as to continue to reduce the performance measure.

Having reviewed the general concepts underlying network topologies and learning methods, let's move into the discussion of a specific neural network, the RBM. As we'll see, the RBM is a key part of a powerful deep learning algorithm.

Restricted Boltzmann Machine

The RBM is a fundamental part of this chapter's subject deep learning architecture—the DBN. The following sections will begin by introducing the theory behind an RBM, including the architectural structure and learning processes.

Following that, we'll dive straight into the code for an RBM class, making links between the theoretical elements and functions in code. We'll finish by touching on the applications of RBMs and the practical factors associated with implementing an RBM.

Introducing the RBM

A Boltzmann machine is a particular type of stochastic, recurrent neural network. It is an energy-based model, which means that it uses an energy function to associate an energy value with each configuration of the network.

We briefly discussed the structure of a Boltzmann machine in the previous section. As mentioned, a Boltzmann machine is a directed cyclic graph, where every node is connected to all other nodes. This property enables it to model in a recurrent fashion, such that the model's outputs evolve and can be viewed over time.

The learning loop in a Boltzmann machine involves maximizing the probability of the training dataset, X. As noted, the specific performance measure used is energy, which is characterized as the negative log of the probability for a dataset X, given a vector of model parameters, Θ. This measure is calculated and used to update the network's weights in such a way as to minimize the free energy in the network.

The Boltzmann machine has seen particular success in processing image data, including photographs, facial features, and handwriting classification contexts.

Unfortunately, the Boltzmann machine is not practical for more challenging ML problems. This is due to the fact that there are challenges with the machine's ability to scale; as the number of nodes increases, the compute time grows exponentially, eventually leaving us in a position where we're unable to compute the free energy of the network.

> For those with an interest in the underlying formal reasoning, this happens because the probability of a data point, *x*, *p(x; Θ)*, must integrate to *1* over all *x*. Achieving this requires that we use a partition function, *Z*, used as a normalizing constant. (*Z* is a constant such that multiplying a non-negative function by *Z* will make the non-negative function integrate to *1* over all inputs; in this case, over all *x*.)
>
> The probability model function is a function of a set of normal distributions. In order to get the energy for our model, we need to differentiate for each of the model's parameters; however, this becomes complicated because of the partition function. Each model parameter produces equations dependent on other model parameters and we ultimately find ourselves unable to calculate the energy without (potentially) hugely expensive calculations, whose cost increases as the network scales.
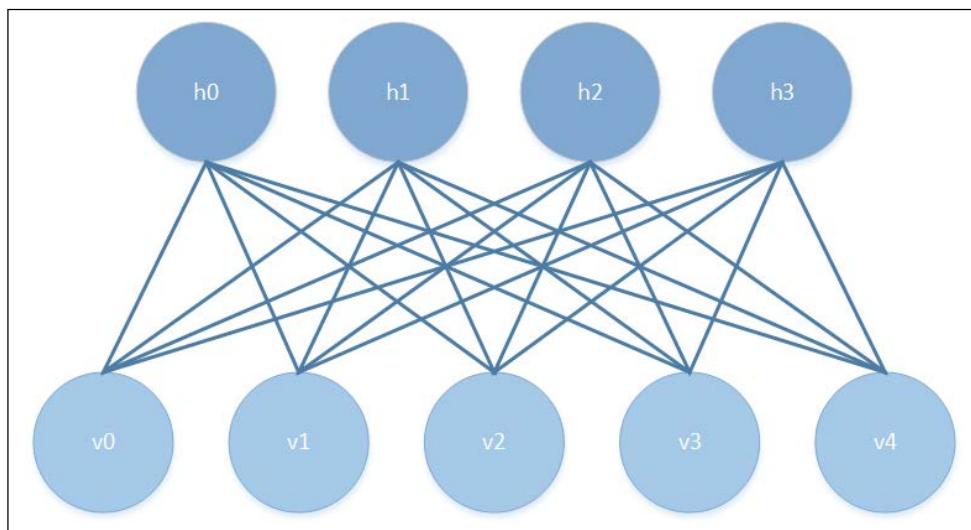
In order to overcome the weaknesses of the Boltzmann machine, it is necessary to make adjustments to both the network topology and training process.

Topology

The main topological change that delivers efficiency improvements is the restriction of connectivity between nodes. First, one must prevent connection between nodes within the same layer. Additionally, all skip-layer connections (that is, direct connections between non-consecutive layers) must be prevented. A Boltzmann machine with this architecture is referred to as an RBM and appears as shown in the following diagram:

One advantage of this topology is that the hidden and visible layers are conditionally independent given one another. As such, it is possible to sample from one layer using the activations of the other.

Training

We observed previously that, for Boltzmann machines, the training time of the machine scales extremely poorly as the machine is scaled up to additional nodes, putting us in a position where we cannot evaluate the energy function that we're attempting to use in training.

The RBM is typically trained using a procedure with a different learning algorithm at its heart, the **Permanent Contrastive Divergence** (**PCD**) algorithm, which provides an approximation of maximum likelihood. PCD doesn't evaluate the energy function itself, but instead allows us to estimate the gradient of the energy function. With this information, we can proceed by making very small adjustments in the direction of the steepest gradient via which we may progress, as desired, toward the local minimum.

The PCD algorithm is made up of two phases. These are referred to as the positive and negative phases, and each phase has a corresponding effect on the energy of the model. The positive phase increases the probability of the training dataset, X, thus reducing the energy of the model. Following this, the negative phase uses a sampling approach from the model to estimate the negative phase gradient. The overall effect of the negative phase is to decrease the probability of samples generated by the model.

Sampling in the negative phase and throughout the update process is achieved using a form of sampling called **Gibbs sampling**.

Gibbs sampling is a variant of the **Markov Chain Monte Carlo** (**MCMC**) family of algorithms, and samples from an approximated multivariate probability distribution. What this means is, rather than using a summed calculation in building our probabilistic model (just as we might do, for instance, when we flip a coin a certain number of times; in such cases, we may sum the number of heads attempts as a proportion of the sum of all attempts), we approximate the value of an integral instead. The subject of how to create a probabilistic model by approximating an integral deserves more time than this book can give it. As such the *Further reading* section of this chapter provides an excellent paper reference. The key points to bear in mind for now (and stripping out a lot of important detail!) are that, instead of summing each case exactly once, we sample based on the (often non-uniform) distribution of the data in question. Gibbs sampling is a probabilistic sampling method for each parameter in a model, based on all of the other parameter values in that model. As soon as a new parameter value is obtained, it is immediately used in sampling calculations for other parameters.

Some of you may be asking at this point why PCD is necessary. Why not use a more familiar method, such as gradient descent with line search? To put it simply, we cannot easily calculate the free energy of our network as this calculation involves an integration across all the network's nodes. We recognized this limitation when we called out the big weakness of the Boltzmann machine—that the compute time grows exponentially as the number of nodes increases, leaving us in a situation where we're trying to minimize a function whose value we cannot calculate!

What PCD provides is a way to estimate the gradient of the energy function. This enables an approximation of the network's free energy, which is fast enough to be viable for application and has shown to be generally accurate. (Refer to the *Further reading* section for a performance comparison.)

As we saw previously, the RBM's probability model function is the joint distribution of our model parameters, making Gibbs sampling appropriate!

The training loop in an initialized RBM involves several steps:

1. We obtain the current iteration's activated hidden layer weight values.

2. We perform the positive phase of PCD, using the state of the Gibbs chain from the previous iteration as input.

3. We perform the negative phase of PCD using the pre-existing state of the Gibbs chain. This gives us the free energy value.

4. We update the activated weights on the hidden layer using the energy value we've calculated.

This algorithm allows the RBM to iteratively step toward a decreased free energy value. The RBM continues to train until both the probability of the training dataset integrates to one and free energy is equal to zero, at which point the RBM has converged.

Now that we've had a chance to review the RBM's topology and training process, let's apply the algorithm to classify a substantial real dataset.

Applications of the RBM

Now that we have a general working knowledge of the RBM algorithm, let's walk through code to create an RBM. We'll be working with an RBM class that will allow us to classify the MNIST handwritten digits dataset. The code we're about to review does the following:

- It sets up the initial parameters of an RBM, including layer size, shareable bias vectors, and shareable weight matrix for connectivity with external network structures (this enables deep belief networks)
- It defines functions for communication and inference between hidden and visible layers
- It defines functions that allow us to update the parameters of network nodes
- It defines functions that handle efficient sampling for the learning process, using PCD-k to accelerate sampling (making it possible to compute in a reasonable frame of time)
- It defines functions that compute the free energy of the model (used to calculate the gradient required for PCD-k updates)
- It identifies the **Psuedo-Likelihood** (**PL**), usable as a log-likelihood proxy to guide the selection of appropriate hyperparameters

Let's begin examining our `RBM` class:

```
class RBM(object):
    def __init__(
        self,
        input=None,
        n_visible=784,
        n_hidden=500,
        w=None,
        hbias=None,
        vbias=None,
        numpy_rng=None,
        theano_rng=None
    ):
```

The first element that we need to build is an RBM constructor, which we can use to define the parameters of the model, such as the number of visible and hidden nodes (n_visible and n_hidden) as well as additional parameters that can be used to adjust how the RBM's inference functions and CD updates are performed.

The W parameter can be used as a pointer to a shared weight matrix. This becomes more relevant when implementing a DBN, as we'll see later in the chapter; in such architectures, the weight matrix needs to be shared between different parts of the network.

The hbias and vbias parameters are used similarly as optional references to shared hidden and visible (respectively) units' bias vectors. Again, these are used in DBNs.

The input parameter enables the RBM to be connected, top-to-tail, to other graph elements. This allows one to, for instance, chain RBMs.

Having set up this constructor, we next need to flesh out each of the preceding parameters:

```
self.n_visible = n_visible
self.n_hidden = n_hidden

if numpy_rng is None:
    numpy_rng = numpy.random.RandomState(1234)

if theano_rng is None:
    theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
```

This is fairly straightforward stuff; we set the visible and hidden nodes for our RBM and set up two random number generators. The theano_rng parameter will be used later in our code to sample from the RBM's hidden units:

```
if W is None:
    initial_W = numpy.asarray(
        numpy_rng.uniform(
            low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            size=(n_visible, n_hidden)
        ),
        dtype=theano.config.floatX
    )
```

This code switches up the data type for `W` so that it can be run over the GPU. Next, we set up shared variables using `theano.shared`, which allows a variable's storage to be shared between functions that it appears in. Within the current example, the shared variables that we create will be the weight vector (`W`) and bias variables for hidden and visible units (`hbias` and `vbias`, respectively). When we move on to creating deep networks with multiple components, the following code will allow us to share components between parts of our networks:

```
W = theano.shared(value=initial_W, name='W', borrow=True)

if hbias is None:
    hbias = theano.shared(
        value=numpy.zeros(
            n_hidden,
            dtype=theano.config.floatX
        ),
        name='hbias',
        borrow=True
    )

if vbias is None:
    vbias = theano.shared(
        value=numpy.zeros(
            n_visible,
            dtype=theano.config.floatX
        ),
        name='vbias',
        borrow=True
    )
```

At this point, we're ready to initialize the input layer as follows:

```
self.input = input
if not input:
    self.input = T.matrix('input')

self.W = W
self.hbias = hbias
self.vbias = vbias
self.theano_rng = theano_rng
self.params = [self.W, self.hbias, self.vbias]
```

As we now have an initialized `input` layer, our next task is to create the symbolic graph that we described earlier in the chapter. Achieving this is a matter of creating functions to manage the interlayer propagation and activation computation operations of the network:

```
def propup(self, vis):
        pre_sigmoid_activation = T.dot(vis, self.W) + self.hbias
        return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_
activation)]


    def propdown(self, hid):
        pre_sigmoid_activation = T.dot(hid, self.W.T) + self.vbias
        return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_
activation)]
```

These two functions pass the activation of one layer's units to the other layer. The first function passes the visible units' activation upward to the hidden units so that the hidden units can compute their activation conditional on a sample of the visible units. The second function does the reverse—propagating the hidden layer's activation downward to the visible units.

It's probably worth asking why we're creating both `propup` and `propdown`. As we reviewed it, PCD only requires that we perform sampling from the hidden units. So what's the value of `propup`?

In a nutshell, sampling from the visible layer becomes useful when we want to sample from the RBM to review its progress. In most applications where our RBM is processing visual data, it is immediately valuable to periodically take the output of sampling from the visible layer and plot it, as shown in the following example:

As we can see here, over the course of iteration, our network begins to change its labeling; in the first case, 7 morphs into 9, while elsewhere 9 becomes 6 and the network gradually reaches a definition of 3-ness.

As we discussed earlier, it's helpful to have as many views on the operation of your RBM as possible to ensure that it's delivering meaningful results. Sampling from the outputs it generates is one way to improve this visibility.

Armed with information about the visible layer's activation, we can deliver a sample of the unit activations from the hidden layer, given the activation of the hidden nodes:

```
    def sample_h_given_v(self, v0_sample):


  pre_sigmoid_h1, h1_mean = self.propup(v0_sample)
    h1_sample = self.theano_rng.binomial(size=h1_mean.shape,
    n=1, p=h1_mean, dtype=theano.config.floatX)


    return [pre_sigmoid_h1, h1_mean, h1_sample]
```

Likewise, we can now sample from the visible layer given hidden unit activation information:

```
    def sample_v_given_h(self, h0_sample):
    pre_sigmoid_v1, v1_mean = self.propdown(h0_sample)
      v1_sample = self.theano_rng.binomial(size=v1_mean.shape,
     n=1, p=v1_mean, dtype=theano.config.floatX)


    return [pre_sigmoid_v1, v1_mean, v1_sample]
```

We've now achieved the connectivity and update loop required to perform a Gibbs sampling step, as described earlier in this chapter. Next, we should define this sampling step!

```
    def gibbs_hvh(self, h0_sample):

        pre_sigmoid_v1, v1_mean, v1_sample =
        self.sample_v_given_h(h0_sample)
        pre_sigmoid_h1, h1_mean, h1_sample =
        self.sample_h_given_v(v1_sample)
        return [pre_sigmoid_v1, v1_mean, v1_sample,
                pre_sigmoid_h1, h1_mean, h1_sample]
```

As discussed, we need a similar function to sample from the visible layer:

```
def gibbs_vhv(self, v0_sample):

    pre_sigmoid_h1, h1_mean, h1_sample =
    self.sample_h_given_v(v0_sample)
    pre_sigmoid_v1, v1_mean, v1_sample =
    self.sample_v_given_h(h1_sample)
    return [pre_sigmoid_h1, h1_mean, h1_sample,
            pre_sigmoid_v1, v1_mean, v1_sample]
```

The code that we've written so far gives us some of our model. It set up the nodes and layers and connections between layers. We've written the code that we need in order to update the network based on Gibbs sampling from the hidden layer.

What we're still missing is code that allows us to perform the following:

- Compute the free energy of the model. As we discussed, the model uses energy as the term to do the following:
  - ° Implement PCD using our Gibbs sampling step code, and setting the Gibbs step count parameter, *k = 1*, to compute the parameter gradient for gradient descent
  - ° Create a means to feed the output of PCD (the computed gradient) to our previously defined network update code

- Develop the means to track the progress and success of our RBM throughout the training.

First off, we'll create the means to calculate the free energy of our RBM. Note that this is the inverse log of the probability distribution for the hidden layer, which we discussed earlier:

```
def free_energy(self, v_sample):

    wx_b = T.dot(v_sample, self.W) + self.hbias
    vbias_term = T.dot(v_sample, self.vbias)
    hidden_term = T.sum(T.log(1 + T.exp(wx_b)), axis=1)
    return -hidden_term - vbias_term
```

Next, we'll implement PCD. At this point, we'll be setting a couple of interesting parameters. The `lr`, short for learning rate, is an adjustable parameter used to adjust learning speed. The `k` parameter points to the number of steps to be performed by PCD (remember the PCD-k notation from earlier in the chapter?).

We discussed the PCD as containing two phases, positive and negative. The following code computes the positive phase of PCD:

```
def get_cost_updates(self, lr=0.1, persistent = , k=1):

        pre_sigmoid_ph, ph_mean, ph_sample =
        self.sample_h_given_v(self.input)



        chain_start = persistent
```

Meanwhile, the following code implements the negative phase of PCD. To do so, we scan the `gibbs_hvh` function `k` times, using Theano's scan operation, performing one Gibbs sampling step with each scan. After completing the negative phase, we acquire the free energy value:

```
        (
            [
                pre_sigmoid_nvs,
                nv_means,
                nv_samples,
                pre_sigmoid_nhs,
                nh_means,
                nh_samples
            ],
            updates
        ) = theano.scan(
            self.gibbs_hvh,
            outputs_info=[None, None, None, None, None, chain_start],
            n_steps=k
        )


        chain_end = nv_samples[-1]

        cost = T.mean(self.free_energy(self.input)) - T.mean(
            self.free_energy(chain_end))

        gparams = T.grad(cost, self.params,
        consider_constant=[chain_end])
```

Having written code that performs the full PCD process, we need a way to feed the outputs to our network. At this point, we're able to connect our PCD learning process to the code to update the network that we reviewed earlier. The preceding updates dictionary points to `theano.scan` of the `gibbs_hvh` function. As you may recall, `gibbs_hvh` currently contains rules for random states of `theano_rng`. What we need to do now is add the new parameter values and variable containing the state of the Gibbs chain to the dictionary (the `updates` variable):

```
for gparam, param in zip(gparams, self.params):
    updates[param] = param - gparam * T.cast(
        lr,
        dtype=theano.config.floatX
    )

    updates = nh_samples[-1]
    monitoring_cost =
    self.get_pseudo_likelihood_cost(updates)


    return monitoring_cost, updates
```

We now have almost all the parts that we need to make our RBM work. What's clearly missing is a means to inspect training, either during or after completion, to ensure that our RBM is learning an appropriate representation of the data.

We talked previously about how to train an RBM, specifically about challenges posed by the partition function. Furthermore, earlier in the code, we implemented one means by which we can inspect an RBM during training; we created the `gibbs_vhv` function to perform Gibbs sampling from the model.

In our previous discussion around how to validate an RBM, we discussed visually plotting the filters that the RBM has created. We'll review how this can be achieved shortly.

The final possibility is to use the inverse log of the PL as a more tractable proxy to the likelihood itself. Technically, the log-PL is the sum of the log-probabilities of each data point (each x) conditioned on all other data points. As discussed, this becomes too expensive with larger-dimensional datasets, so a stochastic approximation to log-PL is used.

We referenced a function that will enable us to get PL cost during the `get_cost_` `updates` function, specifically the `get_pseudo_likelihood_cost` function. Now it's time to flesh out this function and obtain the pseudo-likelihood:

```
def get_pseudo_likelihood_cost(self, updates):

        bit_i_idx = theano.shared(value=0, name='bit_i_idx')
        xi = T.round(self.input)

        fe_xi = self.free_energy(xi)

        xi_flip = T.set_subtensor(xi[:, bit_i_idx], 1 - xi[:,
        bit_i_idx])

        fe_xi_flip = self.free_energy(xi_flip)

        cost = T.mean(self.n_visible *
        T.log(T.nnet.sigmoid(fe_xi_flip - fe_xi)))


        updates[bit_i_idx] = (bit_i_idx + 1) % self.n_visible

        return cost
```

We've now filled out each element on the list of missing components and have completely reviewed the RBM class. We've explored how each element ties into the theory behind the RBM and should now have a thorough understanding of how the RBM algorithm works. We understand what the outputs of our RBM will be and will soon be able to review and assess them. In short, we're ready to train our RBM. Beginning the training of the RBM is a matter of running the following code, which triggers the `train_set_x` function. We'll discuss this function in greater depth later in this chapter:
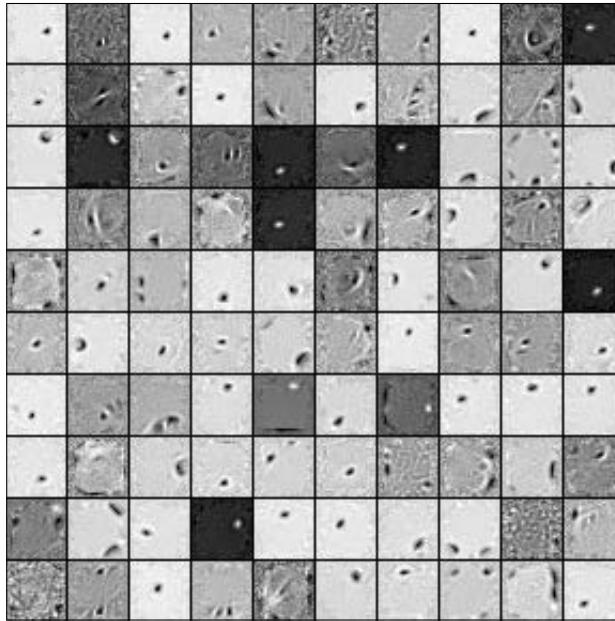
```
    train_rbm = theano.function(
        [index],
        cost,
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) *
            batch_size]
        },
        name='train_rbm'
    )

    plotting_time = 0.
    start_time = time.clock()
```

Having updated the RBM's updates and training set, we run through training epochs. Within each epoch, we train over the training data before plotting the weights as a matrix (as described earlier in the chapter):

```
for epoch in xrange(training_epochs):

    mean_cost = []
    for batch_index in xrange(n_train_batches):
        mean_cost += [train_rbm(batch_index)]

    print 'Training epoch %d, cost is ' % epoch,
    numpy.mean(mean_cost)

    plotting_start = time.clock()
    image = Image.fromarray(
        tile_raster_images(
            X=rbm.W.get_value(borrow=True).T,
            img_shape=(28, 28),
            tile_shape=(10, 10),
            tile_spacing=(1, 1)
        )
    )
    image.save('filters_at_epoch_%i.png' % epoch)
    plotting_stop = time.clock()
    plotting_time += (plotting_stop - plotting_start)

end_time = time.clock()

pretraining_time = (end_time - start_time) - plotting_time

print ('Training took %f minutes' % (pretraining_time / 60.))
```

The weights tend to plot fairly recognizably and resemble Gabor filters (linear filters commonly used for edge detection in images). If your dataset is handwritten characters on a fairly low-noise background, you tend to find that the weights trace the strokes used. For photographs, the filters will approximately trace edges in the image. The following image shows an example output:

Finally, we create the persistent Gibbs chains that we need to derive our samples. The following function performs a single Gibbs step, as discussed previously, then updates the chain:

```
plot_every = 1000

    (
        [
            presig_hids,
            hid_mfs,
            hid_samples,
            presig_vis,
            vis_mfs,
            vis_samples
        ],
        updates
    ) = theano.scan(
        rbm.gibbs_vhv,
        outputs_info=[None, None, None, None, None, persistent_vis_
chain],
        n_steps=plot_every
    )
```

This code runs the `gibbs_vhv` function we described previously, plotting network output samples for our inspection:

```
updates.update({persistent_vis_chain: vis_samples[-1]})
sample_fn = theano.function(
    [],
    [
        vis_mfs[-1],
        vis_samples[-1]
    ],
    updates=updates,
    name='sample_fn'
)

image_data = numpy.zeros(
    (29 * n_samples + 1, 29 * n_chains - 1),
    dtype='uint8'
)
for idx in xrange(n_samples):

    vis_mf, vis_sample = sample_fn()
    print ' ... plotting sample ', idx
    image_data[29 * idx:29 * idx + 28, :] = tile_raster_images(
        X=vis_mf,
        img_shape=(28, 28),
        tile_shape=(1, n_chains),
        tile_spacing=(1, 1)
    )

    image = Image.fromarray(image_data)
    image.save('samples.png')
```

At this point, we have an entire RBM. We have the PCD algorithm and the ability to update the network using this algorithm and Gibbs sampling. We have several visible output methods so that we can assess how well our RBM has trained.

However, we're not done yet! Next, we'll begin to see what the most frequent and powerful application of the RBM is.

Further applications of the RBM

We can use the RBM as an ML algorithm in and of itself. It functions comparably well with other algorithms. Advantageously, it can be scaled up to a point where it can learn high-dimensional datasets. However, this isn't where the real strength of the RBM lies.

The RBM is most commonly used as a pretraining mechanism for a highly effective deep network architecture called a DBN. DBNs are extremely powerful tools to learn and classify a range of image datasets. They possess a very good ability to generalize to unknown cases and are among the best image-learning tools available. For this reason, DBNs are in use at many of the world's top tech and data science companies, primarily in image search and recognition contexts.

Deep belief networks

A DBN is a graphical model, constructed using multiple stacked RBMs. While the first RBM trains a layer of features based on input from the pixels of the training data, subsequent layers treat the activations of preceding layers as if they were pixels and attempt to learn the features in subsequent hidden layers. This is frequently described as learning the representation of data and is a common theme in deep learning.

How many multiple RBMs there should be depends on what is needed for the problem at hand. From a practical perspective, it's a trade-off between increasing accuracy and increasing computational cost. It is the case that each layer of RBMs will improve the lower bound of the log probability of the training data. In other words; the DBN almost inevitably becomes less bad with each additional layer of features.

As far as layer size is concerned, it is generally advantageous to reduce the number of nodes in the hidden layers of successive RBMs. One should avoid contexts in which an RBM has at least as many visible units as the RBM preceding it has hidden units (which raises the risk of simply learning the identity function of the network).

It can be advantageous (but is by no means necessary) when successive RBMs decrease in layer size until the final RBM has a layer size approximating the dimensionality of variance in the data. Affixing an MLP to the end of a DBN whose layers have too many nodes will harm classification performance; it's like trying to affix a drinking straw to the end of a hosepipe! Even an MLP with many neurons may not successfully train in such contexts. On a related note, it has been noted that even if the layers don't contain very many nodes, with enough layers, more or less any function can be modeled.

Determining what the dimensionality of variance in the data is, is not a simple task. One tool that can support this task is PCA; as we saw in the preceding chapter, PCA can enable us to get a reasonable idea as to how many components of meaningful size exist in the input data.

Training a DBN

Training a DBN is typically done greedily, which is to say that it trains to optimize locally at each layer, rather than attempting to reach a global optimum. The learning process is as follows:

- The first layer of the DBN is trained using the method that we saw in our earlier discussion of RBM learning. As such, the first layer converts its data distribution to a posterior distribution using Gibbs sampling over the hidden units.

- This distribution is far more conducive for RBM training than the input data itself so the next RBM layer learns that distribution!

- Successive RBM layers continue to train on the samples output by preceding layers.

- All of the parameters within this architecture are tuned using a performance measure.

This performance measure may vary. It may be a log-likelihood proxy used in gradient descent, as discussed earlier in the chapter. In supervised contexts, a classifier (for example, an MLP) can be added as the final layer of the architecture and prediction accuracy can be used as the performance measure to fine-tune the deep architecture.

Let's move on to using the DBN in practice.

Applying the DBN

Having discussed the DBN and theory surrounding it, it's time to set up our own. We'll be working in a similar way to the RBM, by walking through a DBN class and connecting the code to the theory, discussing what to expect and how to review the network's performance, before initializing and training our network to see it in action.

Let's take a look at our DBN class:

```
class DBN(object):

    def __init__(self, numpy_rng, theano_rng=None, n_ins=784,
```

```
                hidden_layers_sizes=[500, 500], n_outs=10):

        self.sigmoid_layers = []
        self.rbm_layers = []
        self.params = []
        self.n_layers = len(hidden_layers_sizes)

        assert self.n_layers > 0

        if not theano_rng:
            theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))


        self.x = T.matrix('x')
        self.y = T.ivector('y')
```

The `DBN` class contains a number of parameters that bear further explanation. The `numpy_rng` and `theano_rng` parameters, used to determine initial weights, are already familiar from our examination of the `RBM` class. The `n_ins` parameter is a pointer to the dimension (in features) of the DBN's input. The `hidden_layers_sizes` parameter is a list of hidden layer sizes. Each value in this list will guide the `DBN` constructor in creating an RBM layer of the relevant size; as you'll note, the `n_layers` parameter refers to the number of layers in the network and is set by `hidden_layers_sizes`. Adjustment of values in this list enables us to make DBNs whose layer sizes taper down from the input layer size, to increasingly succinct representations, as discussed earlier in the chapter.

It's also worth noting that `self.sigmoid_layers` will store the MLP component (the final layer of the DBN), while `self.rbm_layers` stores the RBM layers used to pretrain the MLP.

With this done, we do the following to complete our DBN architecture:

- We create `n_layers` sigmoid layers
- We connect the sigmoid layers to form an MLP
- We construct an RBM for each sigmoid layer with a shared weight matrix and hidden bias between each sigmoid layer and RBM

The following code creates `n_layers` many layers with sigmoid activations; first creating the input layer, then creating hidden layers whose size corresponds to the values in our `hidden_layers_sizes` list:

```
        for i in xrange(self.n_layers):

            if i == 0:
```

```
                input_size = n_ins
        else:
                    input_size = hidden_layers_sizes[i - 1]
        if i == 0:
            layer_input = self.x
        else:
            layer_input = self.sigmoid_layers[-1].output

        sigmoid_layer = HiddenLayer(rng=numpy_rng,
                                    input=layer_input,
                                    n_in=input_size,
                                    n_out=hidden_layers_sizes[i],
                                    activation=T.nnet.sigmoid)
        self.sigmoid_layers.append(sigmoid_layer)

        self.params.extend(sigmoid_layer.params)
```

Next up, we create an RBM that shares weights with the sigmoid layers. This directly leverages the RBM class that we described previously:

```
        rbm_layer = RBM(numpy_rng=numpy_rng,
                        theano_rng=theano_rng,
                        input=layer_input,
                        n_visible=input_size,
                        n_hidden=hidden_layers_sizes[i],
                        W=sigmoid_layer.W,
                        hbias=sigmoid_layer.b)
        self.rbm_layers.append(rbm_layer)
```

Finally, we add a logistic regression layer to the end of the DBN so as to form an MLP:

```
        self.logLayer = LogisticRegression(
            input=self.sigmoid_layers[-1].output,
            n_in=hidden_layers_sizes[-1],
            n_out=n_outs)
        self.params.extend(self.logLayer.params)

        self.finetune_cost = self.logLayer.negative_log_
likelihood(self.y)

        self.errors = self.logLayer.errors(self.y)
```

Now that we've put together our `MLP` class, let's construct `DBN`. The following code constructs the network with `28 * 28` inputs (that is, `28*28` pixels in the MNIST image data), three hidden layers of decreasing size, and `10` output values (for each of the `10` handwritten number classes in the MNIST dataset):

```
numpy_rng = numpy.random.RandomState(123)
print '... building the model'
dbn = DBN(numpy_rng=numpy_rng, n_ins=28 * 28,
          hidden_layers_sizes=[1000, 800, 720],
          n_outs=10)
```

As discussed earlier in this section, a DBN trains in two stages—a layer-wise pretraining in which each layer takes the output of the preceding layer to train on, which is followed by a fine-tuning step (backpropagation) that allows for weight adjustment across the whole network. The first stage, pretraining, is achieved by performing one step of PCD within each layer's RBM. The following code will perform this pretraining step:

```
print '... getting the pretraining functions'
pretraining_fns =
dbn.pretraining_functions(train_set_x=train_set_x,
batch_size=batch_size, k=k)

print '... pre-training the model'
start_time = time.clock()

for i in xrange(dbn.n_layers):
    for epoch in xrange(pretraining_epochs):
        c = []
        for batch_index in xrange(n_train_batches):
            c.append(pretraining_fns[i](index=batch_index,
                                        lr=pretrain_lr))
        print 'Pre-training layer %i, epoch %d, cost ' % (i,
epoch),
        print numpy.mean(c)

end_time = time.clock()
```

Running the pretrained DBN is then achieved by the following command:

python code/DBN.py

> Note that even with GPU acceleration, this code will spend quite a lot of time pretraining, and it is therefore suggested that you run it overnight.

Validating the DBN

Validation of a DBN as a whole is done in a very familiar way. We can use the minimal validation error from cross-validation as one error measure. However, the minimal cross-validation error can underestimate the error expected on cross-validation data as the meta-parameters may overfit to the new data.

As such, we should use our cross-validation error to adjust our metaparameters until the cross-validation error is minimized. Then we should expose our DBN to the held-out test set, using test error as our validation measure. Our DBN class performs exactly this training process.

However, this doesn't tell us exactly what to do if the network fails to train adequately. What do we do if our DBN is underperforming?

The first thing to do is recognize the potential causes and, in this area, there are some usual culprits. We know that the training of underlying RBMs is also quite tricky and any individual layer may fail to train. Thankfully, our RBM class gives us the ability to tap into and view the weights (filters) being generated by each layer, and we can plot these to get a view on what our network is attempting to represent.

Additionally, we want to ask whether our network is overfitting, or else, underfitting. Either is entirely possible and it's useful to recognize how and why this might be happening. In the case of underfitting, the training process may simply be unable to find good parameters for the model. This is particularly common when you are using a larger network to resolve a large problem space, but can be seen even with some smaller models. If you think that underfitting might be happening with your DBN, you have a couple of options. The first is to simply reduce the size of your hidden layers. This may, or may not, work well. A better alternative is to gradually taper your hidden layers such that each layer learns a refined version of the preceding layer's representation. How to do this, how sharply to taper, and when to stop is a matter of trial and error in the first case and of experience-based learning over the long term.

Overfitting is a well-known phenomenon where your algorithm trains overly specifically on the training data provided. This class of problem is typically identified at the point of cross-validation (where your error rate will increase dramatically), but can be quite pernicious. Means of resolving an overfitting issue do exist; one can increase the training dataset size. A more heavy-handed Bayesian approach would be to attach an additional criterion (for example, a prior) that is used to reduce the value of fitting the training data. Some of the most effective methods to improve classification performance are preprocessing methods, which we'll discuss in *Chapters 6*, *Text Feature Engineering* and *Chapter 7*, *Feature Engineering Part II*.

Though this code will initialize from a predefined position (given a seed value), the stochastic nature of the model means that it will quickly diverge and results may vary. When running on my system, this DBN achieved a minimal cross-validation error of 1.19%. More importantly, it achieved a test error of 1.30% after 46 supervised epochs. These are good results; indeed, they are comparable with field-leading examples!

Further reading

For a primer on neural networks, it makes sense to read from a range of sources. There are many concerns to be aware of and different authors emphasize on different material. A solid introduction is provided by Kevin Gurney in An Introduction to Neural Networks.

An excellent piece on the intuitions underlying Markov Chain Monte Carlo is available at `http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/`.

For readers with a specific interest in the intuitions supporting Gibbs Sampling, Philip Resnik, and Eric Hardisty's paper, *Gibbs Sampling for the Uninitiated*, provides a technical, but clear description of how Gibbs works. It's particularly notable to have some really first-rate analogies! Find them at `https://www.umiacs.umd.edu/~resnik/pubs/LAMP-TR-153.pdf`.

There aren't many good explanations of Contrastive Divergence, one I like is provided by Oliver Woodford at `http://www.robots.ox.ac.uk/~ojw/files/NotesOnCD.pdf`. If you're a little daunted by the heavy use of formal expressions, I would still recommend that you read it for its articulate description of theory and practical concerns involved.

This chapter used the Theano documentation available at `http://deeplearning.net/tutorial/contents.html` as a base for discussion and implementation of RBM and DBN classes.

Summary

We've covered a lot of ground in this chapter! We began with an overview of Neural Networks, focusing on the general properties of topology and learning method before taking a deep dive into the RBM algorithm and RBM code itself. We took this solid understanding forward to create a DBN. In doing so, we linked the DBN theory and code together, before firing up our DBN to work over the MNIST dataset. We performed image classification in a 10-class problem and achieved an extremely competitive result, with classification error below 2%!

In the next chapter, we'll continue to build on your mastery of deep learning by introducing you to another deep learning architecture—**Stacked Denoising Autoencoders** (**SDA**).

3
Stacked Denoising Autoencoders

In this chapter, we'll continue building our skill with deep architectures by applying **Stacked Denoising Autoencoders** (**SdA**) to learn feature representations for high-dimensional input data.

We'll start, as before, by gaining a solid understanding of the theory and concepts that underpin autoencoders. We'll identify related techniques and call out the strengths of autoencoders as part of your data science toolkit. We'll discuss the use of **Denoising Autoencoders** (**dA**), a variation of the algorithm that introduces stochastic corruption to the input data, obliging the autoencoder to decorrupt the input and, in so doing, build a more effective feature representation.

We'll follow up on theory, as before, by walking through the code for a dA class, linking theory and implementation details to build a strong understanding of the technique.

At this point, we'll take a journey very similar to that taken in the preceding chapter—by stacking dA, we'll create a deep architecture that can be used to pretrain an MLP network, which offers substantial performance improvements in a range of unsupervised learning applications including speech data processing.

Autoencoders

The autoencoder (also called the **Diabolo network**) is another crucial component of deep architectures. The autoencoder is related to the RBM, with autoencoder training resembling RBM training; however, autoencoders can be easier to train than RBMs with contrastive divergence and are thus preferred in contexts where RBMs train less effectively.

Introducing the autoencoder

An autoencoder is a simple three-layer neural network whose output units are directly connected back to the input units. The objective of the autoencoder is to encode the **i-dimensional** input into an **h-dimensional** representation, where $h < i$, before reconstructing (decoding) the input at the output layer. The training process involves iteration over this process until the reconstruction error is minimized—at which point one should have arrived at the most efficient representation of input data (should, barring the possibility of arriving at local minima!).

In a preceding chapter, we discussed PCA as being a powerful dimensionality reduction technique. This description of autoencoders as finding the most efficient reduced-dimensional representation of input data will no doubt be familiar and you may be asking why we're exploring another technique that fulfils the same role.

The simple answer is that like the SOM, autoencoders can provide nonlinear reductions, which enables them to process high-dimensional input data more effectively than PCA. This revives a form of our earlier question—why discuss autoencoders if they deliver what an SOM does, without even providing the illuminating visual presentation?
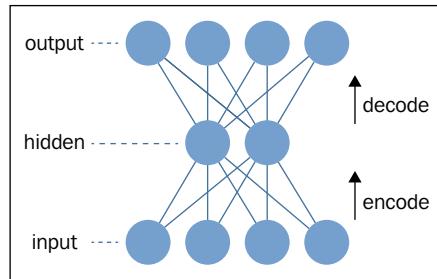
Simply put, autoencoders are a more developed and sophisticated set of techniques; the use of denoising and stacking techniques enable reductions of high-dimensional, multimodal data that can be trained with relative ease to greater accuracy, at greater scale, than the techniques that we discussed in *Chapter 1*, *Unsupervised Machine Learning*.

Having discussed the capabilities of autoencoders at a high level, let's dig in a little further to understand the topology of autoencoders as well as what their training involves.

Topology

As described earlier in this chapter, an autoencoder has a relatively simple structure. It is a three-layer neural network, with **input**, **hidden**, and **output** layers. The **input** feeds forward into the **hidden** layer, then the **output** layer, as with most neural network architectures. One topological feature worth mentioning is that the **hidden** layer is typically of fewer nodes than the **input** or **output** layers. (However, as intimated previously, the required number of **hidden** nodes is really a function of the complexity of the **input** data; the goal of the **hidden** layer is to bottleneck the information content from the **input** and force the network to identify a representation that captures underlying statistical properties. Representing very complex input accurately might require a large quantity of hidden nodes.)

The key feature of an autoencoder is that the **output** is typically set to be the **input**; the performance measure for an autoencoder is its accuracy in reconstructing the **input** after encoding it within the **hidden** layer. Autoencoder topology tends to take the following form:

The encoding function that occurs between the **input** and **hidden** layers is a mapping of an input (x) to a new form (y). A simple example mapping function might be a nonlinear (in this case sigmoid, s) function of the input as follows:

$$y = s(Wx + b)$$

However, more sophisticated encodings may exist or be developed to accommodate specific subject domains. In this case, of course, W represents the weight values assigned to x and b is an adjustable variable that can be tuned to enable the minimization of reconstruction error.

The autoencoder then decodes to deliver its output. This reconstruction is intended to take the same shape as x and will occur through a similar transformation as follows:

$$z = s(W'y + b')$$

Here, b' and W' are typically also configurable to allow network optimization.

Training

The network trains, as discussed, by minimizing the reconstruction error. One popular method to measure this error is a simple squared error measure, as shown in the following formula:

$$E = \frac{1}{2}\|z - x\|^2$$

However, different and more appropriate error measures exist for cases where the input is in a less generic format (such as a set of bit probabilities).

While the intention is that autoencoders capture the main axes of variation in the input dataset, it is possible for an autoencoder to learn something far less useful—the identity function of the input.

Denoising autoencoders

While autoencoders can work well in some applications, they can be challenging to apply to problems where the input data contains a complex distribution that must be modeled in high dimensionality. The major challenge is that, with autoencoders that have **n-dimensional** input and an encoding of at least *n*, there is a real likelihood that the autoencoder will just learn the identity function of the input. In such cases, the encoding is a literal copy of the input. Such autoencoders are called **overcomplete**.

> One of the most important properties when training a machine learning technique is to understand how the dimensionality of hidden layers affects the quality of the resulting model. In cases where the input data is complex and the hidden layer has too few nodes to capture that complexity effectively, the result is obvious—the network fails to train as well as it might with more nodes.
>
> To capture complex distributions in input data, then, you may wish to use a large number of hidden nodes. In cases where the hidden layer has at least as many nodes as the input, there is a strong possibility that the network will learn the identity of the input; in such cases, each element of the input is learned as a specific unique case. Naturally, a model that has been trained to do this will work very well over training data, but as it has learned a trivial pattern that cannot be generalized to unfamiliar data, it is liable to fail catastrophically when validated.

This is particularly relevant when modeling complex data, such as speech data. Such data is frequently complex in distribution, so the classification of speech signals requires multimodal encoding and a high-dimensional hidden layer. Of course, this brings an increased risk of the autoencoder (or any of a large number of models as this is not an autoencoder-specific problem) learning the identity function.

While (rather surprisingly) overcomplete autoencoders can and do learn error-minimizing representations under certain configurations (namely, ones in which the first hidden layer needs very small weights so as to force the hidden units into a linear orientation and subsequent weights have large values), such configurations are difficult to optimize for, and it has been desirable to find another way to prevent overcomplete autoencoders from learning the identity function.

There are several different ways that an overcomplete autoencoder can be prevented from learning the identity function while still capturing something useful within its representation. By far, the most popular approach is to introduce noise to the input data and force the autoencoder to train on the noisy data by learning distributions and statistical regularities rather than identity. This can be effectively achieved by multiple methods, including using sparseness constraints or dropout techniques (wherein input values are randomly set to zero).

The process that we'll be using to introduce noise to the input in this chapter is dropout. Via this method, up to half of the inputs are randomly set to zero. To achieve this, we create a stochastic corruption process that operates on our input data:

```
def get_corrupted_input(self, input, corruption_level):

    return self.theano_rng.binomial(size=input.shape, n=1, p=1 -
    corruption_level, dtype=theano.config.floatX) * input
```

In order to accurately model the input data, the autoencoder has to predict the corrupted values from the uncorrupted values, thus learning meaningful statistical properties (that is, distribution).

In addition to preventing an autoencoder from learning the identity values of data, adding a denoising process also tends to produce models that are substantially more robust to input variations or distortion. This proves to be particularly useful for input data that is inherently noisy, such as speech or image data. One commonly recognized advantage of deep learning techniques, mentioned in the preface to this book, is that deep learning algorithms minimize the need for feature engineering. Where many learning algorithms require lengthy and complicated preprocessing of input data (filtering of images or manipulation of audio signals) to reconstruct the denoised input and enable the model to train, a dA can work effectively with minimal preprocessing. This can dramatically decrease the time it takes to train a model over your input data to practical levels of accuracy.

Finally, it's worth observing that an autoencoder that learns the identity function of the input dataset is probably misconfigured in a fundamental way. As the main added value of the autoencoder is to find a lower-dimensional representation of the feature set, an autoencoder that has learned the identity function of the input data may simply have too many nodes. If in doubt, consider reducing the number of nodes in your hidden layer.

Now that we've discussed the topology of an autoencoder—the means by which one might be effectively trained and the role of denoising in improving autoencoder performance—let's review **Theano** code for a dA so as to carry the preceding theory into practice.

Applying a dA

At this point, we're ready to step through the implementation of a dA. Once again, we're leveraging the Theano library to apply a dA class.

Unlike the RBM class that we explored in the previous chapter, the DenoisingAutoencoder is relatively simple and tying the functionality of the dA to the theory and math that we examined earlier in this chapter is relatively simple.

In *Chapter 2*, *Deep Belief Networks*, we applied an RBM class that had a number of elements that, while not necessary for the correct functioning of the RBM in itself, enabled shared parameters within multilayer, deep architectures. The dA class we'll be using possesses similar shared elements that will provide us with the means to build a multilayer autoencoder architecture later in the chapter.

We begin by initializing a dA class. We specify the number of visible units, n_visible, as well as the number of hidden units, n_hidden. We additionally specify variables for the configuration of the input (input) as well as the weights (W) and the hidden and visible bias values (bhid and bvis respectively). The four additional variables enable autoencoders to receive configuration parameters from other elements of a deep architecture:

```
class dA(object):

    def __init__(
        self,
        numpy_rng,
        theano_rng=None,
        input=None,
        n_visible=784,
        n_hidden=500,
        W=None,
        bhid=None,
        bvis=None
    ):

        self.n_visible = n_visible
        self.n_hidden = n_hidden
```

We follow up by initialising the weight and bias variables. We set the weight vector, `W` to an initial value, `initial_W`, which we obtain using random, uniform sampling from the range:

$$-4 * \sqrt{\frac{6.}{(n\_hidden + n\_visible)}} \quad to \quad 4 * \sqrt{\frac{6.}{(n_{hidden} + n_{visible})}}.$$

We then set the visible and hidden bias variables to arrays of zeroes using `numpy.zeros`:

```
if not theano_rng:
    theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))

if not W:
    initial_W = numpy.asarray(
        numpy_rng.uniform(
            low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            size=(n_visible, n_hidden)
        ),
        dtype=theano.config.floatX
    )
    W = theano.shared(value=initial_W, name='W', borrow=True)

if not bvis:
    bvis = theano.shared(
        value=numpy.zeros(
            n_visible,
            dtype=theano.config.floatX
        ),
        borrow=True
    )

if not bhid:
    bhid = theano.shared(
        value=numpy.zeros(
            n_hidden,
            dtype=theano.config.floatX
        ),
        name='b',
        borrow=True
    )
```

Earlier in the chapter, we described how the autoencoder translates between visible and hidden layers via mappings such as $y = s(Wx + b)$. To enable such translation, it is necessary to define `W`, `b`, `W'`, and `b'` in relation to the previously described autoencoder parameters, `bhid`, `bvis`, and `W`. `W'` and `b'` are referred to as `W_prime` and `b_prime` in the following code:

```
self.W = W
self.b = bhid
self.b_prime = bvis
self.W_prime = self.W.T
self.theano_rng = theano_rng
if input is None:
    self.x = T.dmatrix(name='input')
else:
    self.x = input

self.params = [self.W, self.b, self.b_prime]
```

The preceding code sets `b` and `b_prime` to `bhid` and `bvis` respectively, while `W_prime` is set as the transpose of `W`; in other words, the weights are tied. Tied weights are sometimes, but not always, used in autoencoders for several reasons:

- Tying weights improves the quality of results in several contexts (albeit often in contexts where the optimal solution is PCA, which is the solution an autoencoder with tied weights will tend to reach)

- Tying weights improves the memory consumption of the autoencoder by reducing the number of parameters that need be stored

- Most importantly, tied weights provide a regularization effect; they require one less parameter to be optimized (thus one less thing that can go wrong!)

However, in other contexts, it's both common and appropriate to use untied weights. This is true, for instance, in cases where the input data is multimodal and the optimal decoder models a nonlinear set of statistical regularities. In such cases, a linear model, such as PCA, will not effectively model the nonlinear trends and you will tend to obtain better results using untied weights.

Having configured the parameters to our autoencoder, the next step is to define the functions that enable it to learn. Earlier in this chapter, we determined that autoencoders learn effectively by adding noise to input data, then attempting to learn an encoded representation of that input that can in turn be reconstructed into the input. What we need next, then, are functions that deliver this functionality. We begin by corrupting the input data:

```
def get_corrupted_input(self, input, corruption_level):

    return self.theano_rng.binomial(size=input.shape, n=1, p=1 –
    corruption_level, dtype=theano.config.floatX) * input
```

The degree of corruption is configurable using a `corruption_level` parameter; as we recognized earlier, the corruption of the input through dropout typically does not exceed 50% of cases, or 0.5. The function takes a random set of cases, where the number of cases is that proportion of the `input` whose `size` is equal to `corruption_level`. The function produces a corruption vector of *0's* and *1's* equal in length to the input, where a `corruption_level` sized proportion of the vector is *0*. The corrupted input vector is then simply a multiple of the autoencoder's input vector and corruption vector:

```
def get_hidden_values(self, input):
    return T.nnet.sigmoid(T.dot(input, self.W) + self.b)
```

Next, we obtain the hidden values. This is done via code that performs the equation $y = s(Wx + b)$ to obtain y (the hidden values). To get the autoencoder's output (z), we reconstruct the hidden layer via code that uses the previously defined `b_prime` and `W_prime` to perform $z = s(W'y + b')$:

```
defget_reconstructed_input(self, hidden):
    returnT.nnet.sigmoid(T.dot(hidden, self.W_prime) +
    self.b_prime)
```

The final missing piece is the calculation of cost updates. We reviewed one cost function previously, a simple squared error measure: $E = \frac{1}{2}\|z - x\|^2$. Let's use this cost function to calculate our cost updates, based on the input (x) and reconstruction (z):

```
def get_cost_updates(self, corruption_level, learning_rate):

    tilde_x = self.get_corrupted_input(self.x, corruption_level)
    y = self.get_hidden_values(tilde_x)
```

```
    z = self.get_reconstructed_input(y)
    E = (0.5 * (T.z – T.self.x)) ^ 2
    cost = T.mean(E)

    gparams = T.grad(cost, self.params)
    updates = [
        (param, param - learning_rate * gparam)
        for param, gparam in zip(self.params, gparams)
    ]

return (cost, updates)
```

At this point, we have a functional dA! It may be used to model nonlinear properties of input data and can work as an effective tool to learn valid and lower-dimensional representations of input data. However, the real power of autoencoders comes from the properties that they display when stacked together, as the building blocks of a deep architecture.

Stacked Denoising Autoencoders

While autoencoders are valuable tools in themselves, significant accuracy can be obtained by stacking autoencoders to form a deep network. This is achieved by feeding the representation created by the encoder on one layer into the next layer's encoder as the input to that layer.

Stacked denoising autoencoders (**SdAs**) are currently in use in many leading data science teams for sophisticated natural language analyses as well as a hugely broad range of signals, image, and text analysis.

The implementation of a SdA will be very familiar after the previous chapter's discussion of deep belief networks. The SdA is used in much the same way as the RBMs in our deep belief networks were used. Each layer of the deep architecture will have a dA and sigmoid component, with the autoencoder component being used to pretrain the sigmoid network. The performance measure used by a stacked denoising autoencoder is the training set error, with an intensive period of layer-to-layer (layer-wise) pretraining used to gradually align network parameters before a final period of fine-tuning. During fine-tuning, the network is trained using validation and test data, over fewer epochs but with larger update steps. The goal is to have the network converge at the end of the fine-tuning in order to deliver an accurate result.

In addition to delivering on the typical advantages of deep networks (the ability to learn feature representations for complex or high-dimensional datasets, and the ability to train a model without extensive feature engineering), stacked autoencoders have an additional, interesting property.

Correctly configured stacked autoencoders can capture a **hierarchical grouping** of their input data. Successive layers of a stacked denoised autoencoder may learn increasingly high-level features. Where the first layer might learn some first-order features from input data (such as learning edges in a photo image), a second layer may learn some grouping of first-order features (for instance, by learning given configurations of edges that correspond to contours or structural elements in the input image).

There's no golden rule to determine how many layers or how large layers should be for a given problem. The best solution is usually to experiment with these model parameters until you find an optimal point. This experimentation is best done with a hyperparameter optimization technique or genetic algorithm (subjects we'll discuss in later chapters of this book).

Higher layers may learn increasingly high-order configurations, enabling a stacked denoised autoencoder to learn to recognize facial features, alphanumerical characters, or generalized forms of objects (such as a bird). This is what gives SdAs their unique capability to learn very sophisticated, high-level abstractions of their input data.

Autoencoders can be stacked indefinitely, and it has been demonstrated that continuing to stack autoencoders can improve the effectiveness of the deep architecture (with the main constraint becoming compute cost in time). In this chapter, we'll look at stacking three autoencoders to solve a natural language processing challenge.

Applying the SdA

Now that we've had a chance to understand the advantages and power of the SdA as a deep learning architecture, let's test our skills on a real-world dataset.

For this chapter, let's step away from image datasets and work with the **OpinRank Review** dataset, a text dataset of around 259,000 hotel reviews from TripAdvisor—accessible via the UCI machine learning dataset repository. This freely-available dataset provides review scores (as floating point numbers from 1 to 5) and review text for a broad range of hotels; we'll be applying our stacked dA to attempt to identify the scoring of each hotel from its review text.

> We'll be applying our autoencoder to analyze a preprocessed version of this data, which is accessible from the GitHub share accompanying this chapter. We'll be discussing the techniques by which we prepare text data in an upcoming chapter. For the interested reader, the source data is available at `https://archive.ics.uci.edu/ml/datasets/OpinRank+Review+Dataset`.

In order to get started, we're going to need a stacked denoising autoencoder (hereafter `SdA`) class:

```
class SdA(object):


    def __init__(
        self,
        numpy_rng,
        theano_rng=None,
        n_ins=280,
        hidden_layers_sizes=[500, 500],
        n_outs=5,
        corruption_levels=[0.1, 0.1]
    ):
```

As we previously discussed, the `SdA` is created by feeding the encoding from one layer's autoencoder as the input to the subsequent layer. This class supports the configuration of the layer count (reflected in, but not set by, the length of the `hidden_layers_sizes` and `corruption_levels` vectors). It also supports differentiated layer sizes (in nodes) at each layer, which can be set using `hidden_layers_sizes`. As we discussed, the ability to configure successive layers of the autoencoder is critical to developing successful representations.

Next, we need parameters to store the MLP (`self.sigmoid_layers`) and dA (`self.dA_layers`) elements of the `SdA`. In order to specify the depth of our architecture, we use the `self.n_layers` parameter to specify the number of sigmoid and dA layers required:

```
self.sigmoid_layers = []
self.dA_layers = []
self.params = []
self.n_layers = len(hidden_layers_sizes)

assertself.n_layers> 0
```

Next, we need to construct our sigmoid and dA layers. We begin by setting the hidden layer size to be set either from the input vector size or by the activation of the preceding layer. Following this, `sigmoid_layer` and `dA_layer` components are created, with the dA layer drawing from the `dA` class that we discussed earlier in this chapter:

```
for i in xrange(self.n_layers):
    if i == 0:
        input_size = n_ins
    else:
        input_size = hidden_layers_sizes[i - 1]

    if i == 0:
        layer_input = self.x
    else:
        layer_input = self.sigmoid_layers[-1].output

    sigmoid_layer = HiddenLayer(rng=numpy_rng,
    input=layer_input,
    n_in=input_size,
    n_out=hidden_layers_sizes[i],
    activation=T.nnet.sigmoid)

    self.sigmoid_layers.append(sigmoid_layer)
    self.params.extend(sigmoid_layer.params)

    dA_layer = dA(numpy_rng=numpy_rng,
    theano_rng=theano_rng,
    input=layer_input,
    n_visible=input_size,
    n_hidden=hidden_layers_sizes[i],
    W=sigmoid_layer.W,
    bhid=sigmoid_layer.b)

    self.dA_layers.append(dA_layer)
```

Having implemented the layers of our stacked dA, we'll need a final, logistic regression layer to complete the MLP component of the network:

```
self.logLayer = LogisticRegression(
    input=self.sigmoid_layers[-1].output,
    n_in=hidden_layers_sizes[-1],
    n_out=n_outs
)

self.params.extend(self.logLayer.params)
self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)
self.errors = self.logLayer.errors(self.y)
```

This completes the architecture of our SdA. Next up, we need to generate the training functions used by the `SdA` class. Each function will the minibatch index (`index`) as an argument, together with several other elements—the `corruption_level` and `learning_rate` are enabled here so that we can adjust them (for example, gradually increase or decrease them) during training. Additionally, we identify variables that help identify where the batch starts and ends—`batch_begin` and `batch_end`, respectively:

> The ability to dynamically adjust the learning rate is particularly very helpful and may be applied in one of two ways. Once a technique has begun to converge on an appropriate solution, it is very helpful to be able to reduce the learning rate. If you do not do this, you risk creating a situation in which the network oscillates between values located around the optimum without ever hitting it. In some contexts, it can be helpful to tie the learning rate to the network's performance measure. If the error rate is high, it makes sense to make larger adjustments until the error rate begins to decrease!

```
def pretraining_functions(self, train_set_x, batch_size):
    index = T.lscalar('index')
    corruption_level = T.scalar('corruption')
    learning_rate = T.scalar('lr')
    batch_begin = index * batch_size
    batch_end = batch_begin + batch_size

    pretrain_fns = []
    for dA in self.dA_layers:
        cost, updates = dA.get_cost_updates(corruption_level,
          learning_rate)
        fn = theano.function(
            inputs=[
                index,
                theano.Param(corruption_level, default=0.2),
                theano.Param(learning_rate, default=0.1)
            ],
            outputs=cost,
            updates=updates,
            givens={
                self.x: train_set_x[batch_begin: batch_end]
            }
        )
        pretrain_fns.append(fn)

    return pretrain_fns
```

The pretraining functions that we've created takes the minibatch `index` and can optionally take the corruption level or learning rate. It performs one step of pretraining and outputs the cost value and vector of weight updates.

In addition to pretraining, we need to build functions to support the fine-tuning stage, wherein the network is run iteratively over the validation and test data to optimize network parameters. The training function (`train_fn`) seen in the code below implements a single step of fine-tuning. The `valid_score` is a Python function that computes a validation score using the error measure produced by the `SdA` over validation data. Similarly, `test_score` computes the error score over test data.

To get this process off the ground, we first need to set up training, validation, and test datasets. Each stage requires two datasets (set `x` and set `y`) containing the features and class labels, respectively. The required number of minibatches for validation and test is determined, and an index is created to track the batch size (and provide a means of identifying at which entries a batch starts and ends). Training, validation, and testing occurs for each batch and afterward, both `valid_score` and `test_score` are calculated across all batches:

```python
def build_finetune_functions(self, datasets, batch_size,
  learning_rate):

    (train_set_x, train_set_y) = datasets[0]
    (valid_set_x, valid_set_y) = datasets[1]
    (test_set_x, test_set_y) = datasets[2]

    n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
    n_valid_batches /= batch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0]
    n_test_batches /= batch_size

    index = T.lscalar('index')


    gparams = T.grad(self.finetune_cost, self.params)


    updates = [
        (param, param - gparam * learning_rate)
        For param, gparam in zip(self.params, gparams)
    ]

train_fn = theano.function(
    inputs=[index],
```

```
        outputs=self.finetune_cost,
        updates=updates,
        givens={
            self.x: train_set_x[
                index * batch_size: (index + 1) * batch_size
            ],
            self.y: train_set_y[
                index * batch_size: (index + 1) * batch_size
            ]
        },
        name='train'
    )

    test_score_i = theano.function(
        [index],
        self.errors,
        givens={
            self.x: test_set_x[
            index * batch_size: (index + 1) * batch_size
        ],
            self.y: test_set_y[
            index * batch_size: (index + 1) * batch_size
        ]
    },
        name='test'
    )

    valid_score_i = theano.function(
        [index],
        self.errors,
        givens={
            self.x: valid_set_x[
                index * batch_size: (index + 1) * batch_size
            ],
            self.y: valid_set_y[
                index * batch_size: (index + 1) * batch_size
            ]
        },
        name='valid'
    )


    def valid_score():
```

```
    return [valid_score_i(i) for i inxrange(n_valid_batches)]

def test_score():
    return [test_score_i(i) for i inxrange(n_test_batches)]

return train_fn, valid_score, test_score
```

With the training functionality in place, the following code initiates our stacked dA:

```
numpy_rng = numpy.random.RandomState(89677)
print '... building the model'
    sda = SdA(
        numpy_rng=numpy_rng,
        n_ins=280,
        hidden_layers_sizes=[240, 170, 100],
        n_outs=5
    )
```

It should be noted that, at this point, we should be trying an initial configuration of layer sizes to see how we do. In this case, the layer sizes used are the product of some initial testing. As we discussed, training the SdA occurs in two stages. The first is a layer-wise pretraining process that loops over all of the SdA's layers. The second is a process of fine-tuning over validation and test data.

To pretrain the SdA, we provide the required corruption levels to train each layer and iterate over the layers using our previously defined pretraining_fns:

```
print '... getting the pretraining functions'
pretraining_fns = sda.pretraining_functions(train_set_x=train_set_x,
batch_size=batch_size)

print '... pre-training the model'
start_time = time.clock()
corruption_levels = [.1, .2, .2]
for i in xrange(sda.n_layers):

    for epoch in xrange(pretraining_epochs):
        c = []
        for batch_index in xrange(n_train_batches):
            c.append(pretraining_fns[i](index=batch_index,
            corruption=corruption_levels[i],
            lr=pretrain_lr))
```

```
print 'Pre-training layer %i, epoch %d, cost ' % (i, epoch),

print numpy.mean(c)

end_time = time.clock()

print(('The pretraining code for file ' +
os.path.split(__file__)[1] + ' ran for %.2fm' % ((end_time - start_
time) / 60.)), file = sys.stderr)
```

At this point, we're able to initialize our `SdA` class via calling the preceding code stored within this book's GitHub repository: `MasteringMLWithPython/Chapter3/SdA.py`

Assessing SdA performance

The SdA will take a significant length of time to run. With 15 epochs per layer and each layer typically taking an average of 11 minutes, the network will run for around 500 minutes on a modern desktop system with GPU acceleration and a single-threaded GotoBLAS.

On a system without GPU acceleration, the network will take substantially longer to train, and it is recommended that you use the alternative, which runs over a significantly smaller input dataset: `MasteringMLWithPython/Chapter3/SdA_no_blas.py`

The results are of high quality, with a validation error score of 3.22% and test error score of 3.14%. These results are particularly impressive given the ambiguous and sometimes challenging nature of natural language processing applications.

It was noticeable that the network classified more correctly for the 1-star and 5-star rating cases than for the intermediate levels. This is largely due to the ambiguous nature of unpolarized or unemotional language.

Part of the reason that this input data was classifiable was via significant feature engineering. While time-consuming and sometimes problematic, we've seen that well-executed feature engineering combined with an optimized model can deliver an excellent level of accuracy. In *Chapter 6*, *Text Feature Engineering*, we'll be applying the techniques used to prepare this dataset ourselves.

Further reading

A well-informed overview of autoencoders (amongst other subjects) is provided by Quoc V. Le from the Google Brain team. Read about it at `https://cs.stanford.edu/~quocle/tutorial2.pdf`.

This chapter used the Theano documentation available at `http://deeplearning.net/tutorial/contents.html` as a base for discussion as Theano was the main library used in this chapter.

Summary

In this chapter, we introduced the autoencoder, an effective dimensionality reduction technique with some unique applications. We focused on the theory behind the stacked denoised autoencoder, an extension of autoencoders whereby any number of autoencoders are stacked in a deep architecture. We were able to apply the stacked denoised autoencoder to a challenging natural language processing problem and met with great success, delivering highly accurate sentiment analysis of hotel reviews.

In the next chapter, we will discuss supervised deep learning methods, including **Convolutional Neural Networks** (**CNN**).

4

Convolutional Neural Networks

In this chapter, you'll be learning how to apply the convolutional neural network (also referred to as the CNN or convnet), perhaps the best-known deep architecture, via the following steps:

- Taking a look at the convnet's topology and learning processes, including convolutional and pooling layers
- Understanding how we can combine convnet components into successful network architectures
- Using Python code to apply a convnet architecture so as to solve a well-known image classification task

Introducing the CNN

In the field of machine learning, there is an enduring preference for developing structures in code that parallel biological structures. One of the most obvious examples is that of the MLP neural network, whose topology and learning processes are inspired by the neurons of the human brain.

This preference has turned out to be highly efficient; the availability of specialized, optimized biological structures that excel at specific sets of tasks gives us a wealth of templates and clues from which to design and create effective learning models.

The design of convolutional neural networks takes inspiration from the visual cortex—the area of the brain that processes visual input. The visual cortex has several specializations that enable it to effectively process visual data; it contains many receptor cells that detect light in overlapping regions of the visual field. All receptor cells are subject to the same convolution operation, which is to say that they all process their input in the same way. These specializations were incorporated into the design of convnets, making their topology noticeably distinct from that of other neural networks.

It's safe to say that CNN (convnets for short) are underpinning many of the most impactful current advances in artificial intelligence and machine learning. Variants of CNN are applied to some of the most sophisticated visual, linguistic, and problem-solving applications in existence. Some examples include the following:

- Google has developed a range of specialized convnet architectures, including **GoogLeNet**, a 22-layer convnet architecture. In addition, Google's DeepDream program, which became well-known for its overtrained, hallucinogenic imagery, also uses a convolutional neural network.

- Convolutional nets have been taught to play the game **Go** (a long-standing AI challenge), achieving win-rates ranging between 85% and 91% against highly-ranked players.

- Facebook uses convolutional nets in face verification (**DeepFace**).

- Baidu, Microsoft research, IBM, and Twitter are among the many other teams using convnets to tackle the challenges around trying to deliver next-generation intelligent applications.

In recent years, object recognition challenges, such as the 2014 **ImageNet** challenge, have been dominated by winners employing specialized convnet implementations or multiple-model ensembles that combine convnets with other architectures.

While we'll cover how to create and effectively apply ensembles in *Chapter 8*, *Ensemble Methods*, this chapter focuses on the successful application of convolutional neural networks to large-scale visual classification contexts.

Understanding the convnet topology
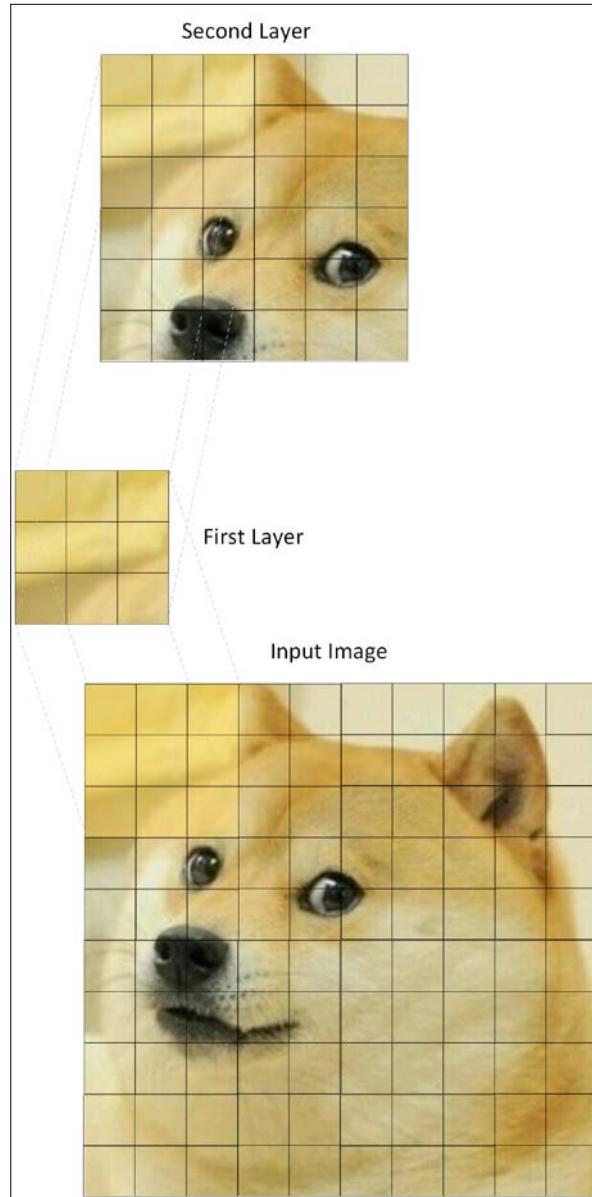
The convolutional neural network's architecture should be fairly familiar; the network is an acyclic graph composed of layers of increasingly few nodes, where each layer feeds into the next. This will be very familiar from many well-known network topologies such as the MLP.

Perhaps the most immediate difference between a convolutional neural network and most other networks is that all of the neurons in a convnet are identical! All neurons possess the same parameters and weight values. As you can see, this will immediately reduce the number of parameter values controlled by the network, bringing substantial efficiency savings. It also typically improves network learning rate as there are fewer free parameters to be managed and computed over. As we'll see later in this chapter, shared weights also enable a convnet to learn features irrespective of their position in the input (for example, the input image or audio signal).

Another big difference between convolutional networks and other architectures is that the connectivity between nodes is limited such as to develop a spatially local connectivity pattern. In other words, the inputs to a given node will be limited to only those nodes whose receptor fields are contiguous. This may be spatially contiguous, as in the case of image data; in such cases, each neuron's inputs will ultimately draw from a continuous subset of the image. In the case of audio signal data, the input might instead be a continuous window of time.

To illustrate this more clearly, let's take an example input image and discuss how a convolutional network might process parts of that image across specific nodes. Nodes in the first layer of a convolutional neural network will be assigned subsets of the input image. In this case, let's say that they take a 3 x 3 pixel subset of the image each. Our coverage covers the entire image without any overlap between the areas taken as input by nodes and without any gaps. (Note that none of these conditions are automatically true for convnet implementations.) Each node is assigned a 3 x 3 pixel subset of the image (the receptive field of the node) and outputs a transformed version of that input. We'll disregard the specifics of that transformation for now.

This output is usually then picked up by a second layer of nodes. In this case, let's say that our second layer is taking a subset of all of the outputs from nodes in the first layer. For example, it might be taking a contiguous 6 x 6 pixel subset of the original image; that is, it has a receptive field that covers the outputs of exactly four nodes from the preceding layer. This becomes a little more intuitive when explained visually:

Each layer is **composable**; the output of one convolutional layer may be fed into the next layer as an input. This provides the same effect that we saw in the *Chapter 3*, *Stacked Denoising Autoencoders*; successive layers develop representations of increasingly high-level, abstract features. Furthermore, as we build downward—adding layers—the representation becomes responsive to a larger region of pixel space. Ultimately, by stacking layers, we can work our way toward global representations of the entire input.

Understanding convolution layers

As described, in order to prevent each node from learning an unpredictable (and difficult to tune!) set of very local, free parameters, weights in a layer are shared across the entire layer. To be completely precise, the filters applied in a convolutional layer are a single set of filters, which are slid (convolved) across the input dataset. This produces a two-dimensional activation map of the input, which is referred to as the feature map.

The filter itself is subject to four hyperparameters: size, depth, stride, and zero-padding. The size of the filter is fairly self-explanatory, being the area of the filter (obviously, found by multiplying height and width; a filter need not be square!). Larger filters will tend to overlap more, and as we'll see, this can improve the accuracy of classification. Crucially, however, increasing the filter size will create increasingly large outputs. As we'll see, managing the size of outputs from convolutional layers is a huge factor in controlling the efficiency of a network.

Depth defines the number of nodes in the layer that connect to the same region of the input. The trick to understanding depth is to recognize that looking at an image (for people or networks) involves processing multiple different types of property. Anyone who has ever looked at all the image adjustment sliders in Photoshop has an idea of what this might entail. Depth is sometimes referred to as a dimension in its own right; it almost relates to the complexity of an image, not in terms of its contents but in terms of the number of channels needed to accurately describe it.
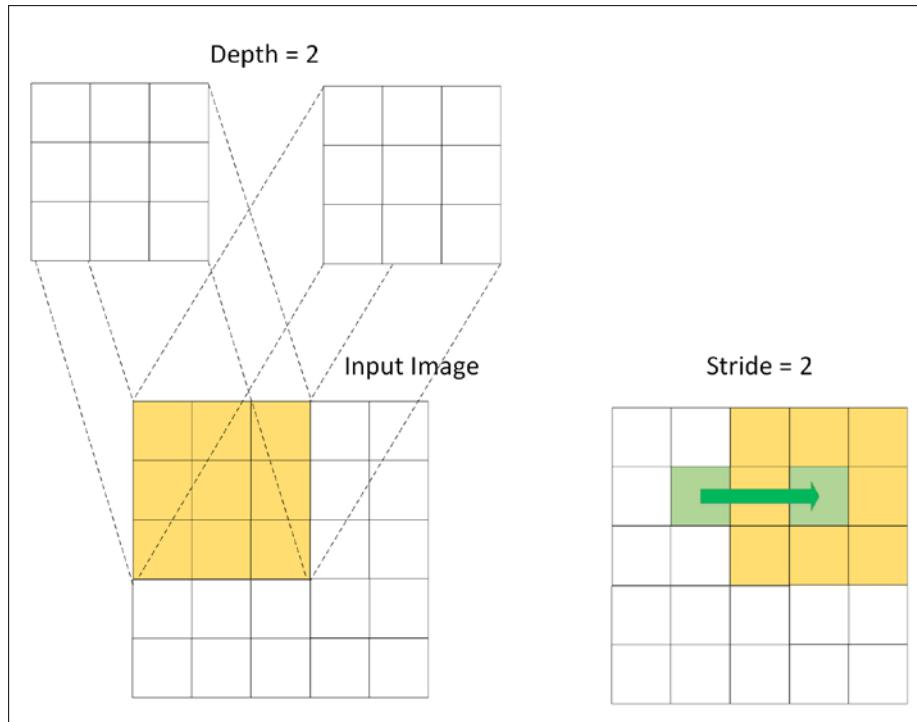
It's possible that the depth might describe color channels, with nodes mapped to recognize green, blue, or red in the input. This, incidentally, leads to a common convention where depth is set to three (particularly at the first convolution layer). It's very important to recognize that some nodes commonly learn to express less easily-described properties of input images that happen to enable a convnet to learn that image more accurately. Increasing the depth hyperparameter tends to enable nodes to encode more information about inputs, with the attendant problems and benefits that you might expect.

As a result, setting the depth parameter to too small a value tends to lead to poor results because the network doesn't have the expressive depth (in terms of channel count) required to accurately characterize input data. This is a problem analogous to not having enough features, except that it's more easily fixed; one can tune the depth of the network upward to improve the expressive depth of the convnet.

Equally, setting the depth parameter to too small a value can be redundant or harmful to performance, thereafter. If in doubt, consider testing the appropriate depth value during network configuration via hyperparameter optimization, the elbow method, or another technique.

Stride is a measure of spacing between neurons. A stride value of one will lead every element of the input (for an image, potentially every pixel) to be the center of a filter instance. This naturally leads to a high degree of overlap and very large outputs. Increasing the stride causes less of an overlap in the receptive fields and the output's size is reduced. While tuning the stride of a convnet is a question of weighing accuracy against output size, it can generally be a good idea to use smaller strides, which tend to work better. In addition, a stride value of one enables us to manage down-sampling and scale reduction at pooling layers (as we'll discuss later in the chapter).

The following diagram graphically displays both **Depth** and **Stride**:

The final hyperparameter, zero-padding, offers an interesting convenience. Zero-padding is the process of setting the outer values (the border) of each receptive field to zero, which has the effect of reducing the output size for that layer. It's possible to set one, or multiple, pixels around the border of the field to zero, which reduces the output size accordingly. There are, of course, limits; obviously, it's not a good idea to set zero-padding and stride such that areas of the input are not touched by a filter! More generally, increasing the degree of zero-padding can cause a decrease in effectiveness, which is tied to the increased difficulty of learning features via coarse coding. (Refer to the *Understanding pooling layers* section in this chapter.)

However, zero-padding is very helpful because it enables us to adjust the input and output sizes to be the same. This is a very common practice; using zero-padding to ensure that the size of the input layer and output layer are equal, we are able to easily manage the stride and depth values. Without using zero-padding in this way, we would need to do a lot of work tracking input sizes and managing network parameters simply to make the network function correctly. In addition, zero-padding also improves performance as, without it, a convnet will tend to gradually degrade content at the edges of the filter.

In order to calibrate the number of nodes, appropriate stride, and padding for successive layers when we define our convnet, we need to know the size of the output from the preceding layer. We can calculate the spatial size of a layer's output (O) as a function of the input image size (W), filter size (F), stride (S), and the amount of zero-padding applied (P), as follows:

$$O = \frac{W - F + 2P}{S + 1}$$

If O is not an integer, the filters do not tile across the input neatly and instead extend over the edge of the input. This can cause some problematic issues when training (normally involving thrown exceptions)! By adjusting the stride value, one can find a whole-number solution for O and train effectively. It is normal for the stride to be constrained to what is possible given the other hyperparameter values and size of the input.

We've discussed the hyperparameters involved in correctly configuring the convolutional layer, but we haven't yet discussed the convolution process itself. Convolution is a mathematical operator, like addition or derivation, which is heavily used in signal processing applications and in many other contexts where its application helps simplify complex equations.

Loosely speaking, convolution is an operation over two functions, such as to produce a third function that is a modified version of one of the two original functions. In the case of convolution within a convnet, the first component is the network's input. In the case of convolution applied to images, convolution is applied in two dimensions (the width and height of the image). The input image is typically three matrices of pixels — one for each of the red, blue, and green color channels, with values ranging between *0* and *255* in each channel.

> At this point, it's worth introducing the concept of a **tensor**. Tensor is a term commonly used to refer to an n-dimensional array or matrix of input data, commonly applied in deep learning contexts. It's effectively analogous to a matrix or array. We'll be discussing tensors in more detail, both in this chapter and in *Chapter 9*, *Additional Python Machine Learning Tools* (where we review the **TensorFlow** library). It's worth noting that the term tensor is noticing a resurgence of use in the machine learning community, largely through the influence of Google machine intelligence research teams.

The second input to the convolution operation is the convolution kernel, a single matrix of floating point numbers that acts as a filter on the input matrices. The output of this convolution operation is the feature map. The convolution operation works by sliding the filter across the input, computing the dot product of the two arguments at each instance, which is written to the feature map. In cases where the stride of the convolutional layer is one, this operation will be performed across each pixel of the input image.

The main advantage of convolution is that it reduces the need for feature engineering. Creating and managing complex kernels and performing the highly specialized feature engineering processes needed is a demanding task, made more challenging by the fact that feature engineering processes that work well in one context can work poorly in most others. While we discuss feature engineering in detail in *Chapter 7*, *Feature Engineering Part II*, convolutional nets offer a powerful alternative.

CNN, however, incrementally improve their kernel's ability to filter a given input, thus automatically optimizing their kernel. This process is accelerated by learning multiple kernels in parallel at once. This is feature learning, which we've encountered in previous chapters. Feature learning can offer tremendous advantages in time and in increasing the accessibility of many problems. As with our earlier SDA and DBN implementations, we would look to pass our learned features to a much simpler, shallow neural network, which uses these features to classify the input image.

Understanding pooling layers

Stacking convolutional layers allows us to create a topology that effectively creates features as feature maps for complex, noisy input data. However, convolutional layers are not the only component of a deep network. It is common to weave convolutional layers in with pooling layers. Pooling is an operation over feature maps, where multiple feature values are aggregated into a single value—mostly using a max (**max-pooling**), mean (**mean-pooling**), or summation (**sum-pooling**) operation.

Pooling is a fairly natural approach that offers substantial advantages. If we do not aggregate feature maps, we tend to find ourselves with a huge amount of features. The **CIFAR-10** dataset that we'll be classifying later in this chapter contains 60,000 32 x 32 pixel images. If we hypothetically learned *200* features for each image—over 8 x 8 inputs—then at each convolution, we'd find ourselves with an output vector of size *(32 – 8+1) * (32 – 8+1) * 200*, or *125,000* features per image. Convolution produces a huge amount of features that tend to make computation very expensive and can also introduce significant overfitting problems.

The other major advantage provided by a pooling operation is that it provides a level of robustness against the many, small deviations and variances that occur in modeling noisy, high-dimensional data. Specifically, pooling prevents the network learning the position of features too specifically (overfitting), which is obviously a critical requirement in image processing and recognition settings. With pooling, the network no longer fixates on the precise location of features in the input and gains a greater ability to generalize. This is called **translation-invariance**.

Max-pooling is the most commonly applied pooling operation. This is because it focuses on the most responsive features in question that should, in theory, make it the best candidate for image recognition and classification purposes. By a similar logic, min-pooling tends to be applied in cases where it is necessary to take additional steps to prevent an overly sensitive classification or overfitting from occurring.

For obvious reasons, it's prudent to begin modeling using a quickly applied and straightforward pooling method such as max-pooling. However, when seeking additional gains in network performance during later iterations, it's important to look at whether your pooling operations can be improved on. There isn't any real restriction in terms of defining your own pooling operation. Indeed, finding a more effective subsampling method or alternative aggregation can substantially improve the performance of your model.

In terms of `theano` code, a max-pooling implementation is pretty straightforward and may look like this:

```
from theano.tensor.signal import downsample

input = T.dtensor4('input')
maxpool_shape = (2, 2)
pool_out = downsample.max_pool_2d(input, maxpool_shape, ignore_
border=True)
f = theano.function([input],pool_out)
```

The `max_pool_2d` function takes an n-dimensional `tensor` and downscaling factor, in this case, `input` and `maxpool_shape`, with the latter being a tuple of length 2, containing width and height downscaling factors for the input image. The `max_pool_2d` operation then performs max-pooling over the two trailing dimensions of the vector:

```
invals = numpy.random.RandomState(1).rand(3, 2, 5, 5)

pool_out = downsample.max_pool_2d(input, maxpool_shape, ignore_
border=False)
f = theano.function([input],pool_out)
```

The `ignore_border` determines whether the border values are considered or discarded. This max-pooling operation produces the following, given that `ignore_border = True`:

```
[[ 0.72032449  0.39676747]
 [ 0.6852195   0.87811744]]
```

As you can see, pooling is a straightforward operation that can provide dramatic results (in this case, the input was a 5 x 5 matrix, reduced to 2 x 2). However, pooling is not without critics. In particular, *Geoffrey Hinton* offered this pretty delightful soundbite:

> "*The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster.*
>
> *If the pools do not overlap, pooling loses valuable information about where things are. We need this information to detect precise relationships between the parts of an object. Its true that if the pools overlap enough, the positions of features will be accurately preserved by "coarse coding" (see my paper on "distributed representations" in 1986 for an explanation of this effect). But I no longer believe that coarse coding is the best way to represent the poses of objects relative to the viewer (by pose I mean position, orientation, and scale).*"

This is a bold statement, but it makes sense. Hinton's telling us that the pooling operation, as an aggregation, does what any aggregation necessarily does—it reduces the data to a simpler and less informationally-rich format. This wouldn't be too damaging, except that Hinton goes further.

Even if we'd reduced the data down to single values for each pool, we could still hope that the fact that multiple pools overlap spatially would still present feature encodings. (This is the coarse coding referred to by Hinton.) This is also quite an intuitive concept. Imagine that you're listening in to a signal on a noisy radio frequency. Even if you only caught one word in three, it's probable that you'd be able to distinguish a distress signal from the shipping forecast!

However, Hinton follows up by observing that coarse coding is not as effective in learning pose (position, orientation, and scale). There are so many permutations in viewpoint relative to an object that it's unlikely two images would be alike and the sheer variety of possible poses becomes a challenge for a convolutional network using pooling. This suggests that an architecture that does not overcome this challenge may not be able to break past an upper limit for image classification.

However, the general consensus, at least for now, is that even after acknowledging all of this, it is still highly advantageous in terms of efficiency and translation-invariance to continue using pooling operations in convnets. Right now, the argument goes that it's the best we have!

Meanwhile, Hinton proposed an alternative to convnets in the form of the **transforming autoencoder**. The transforming autoencoder offers accuracy improvements on learning tasks that require a high level of precision (such as facial recognition), where pooling operations would cause a reduction in precision. The *Further reading* section of this chapter contains recommendations if you are interested in learning more about the transforming autoencoder.

So, we've spent quite a bit of time digging into the convolutional neural network—its components, how they work, and their hyperparameters. Before we move on to put the theory into action, it's worth discussing how all of these theoretical components fit together into a working architecture. To do this, let's discuss what training a convnet looks like.

Training a convnet

The means of training a convolutional network will be familiar to readers of the preceding chapters. The convolutional architecture itself is used to pretrain a simpler network structure (for example, an MLP). The backpropagation algorithm is the standard method to compute the gradient when pretraining. During this process, every layer undertakes three tasks:

- **Forward pass**: Each feature map is computed as a sum of all feature maps convolved with the corresponding weight kernel
- **Backward pass**: The gradients respective to inputs are calculated by convolving the transposed weight kernel with the gradients, with respect to the outputs
- The loss for each kernel is calculated, enabling the individual weight adjustment of every kernel as needed
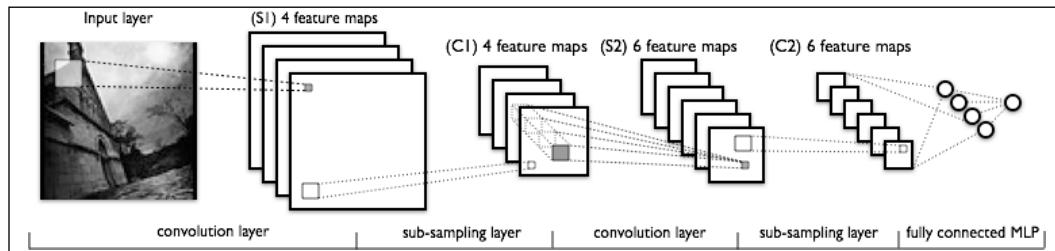
Repetition of this process allows us to achieve increasing kernel performance until we reach a point of convergence. At this point, we will hope to have developed a set of features sufficient that the capping network is able to effectively classify over these features.

This process can execute slowly, even on a fairly advanced GPU. Some recent developments have helped accelerate the training process, including the use of the Fast **Fourier Transform** to accelerate the convolution process (for cases where the convolution kernel is of roughly equal size to the input image).

Putting it all together
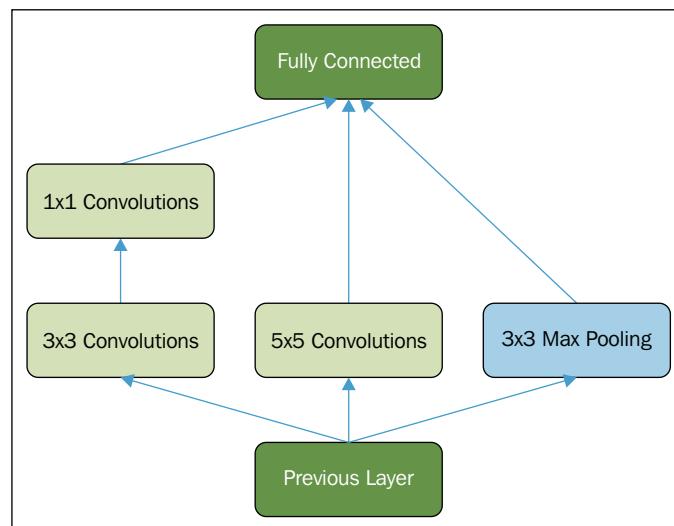
So far, we've discussed some of the elements required to create a CNN. The next subject of discussion should be how we go about combining these components to create capable convolutional nets as well as which combinations of components can work well. We'll draw guidance from a number of forerunning convnet implementations as we build an understanding of what is commonly done as well as what is possible.

Probably the best-known convolutional network implementation is Yann LeCun's **LeNet**. LeNet has gone through several iterations since LeNet-1 in late 1980, but has been increasingly effective at performing tasks including handwritten digit and image classification. LeNet is structured using alternating convolution and pooling layers capped by an MLP, as follows:
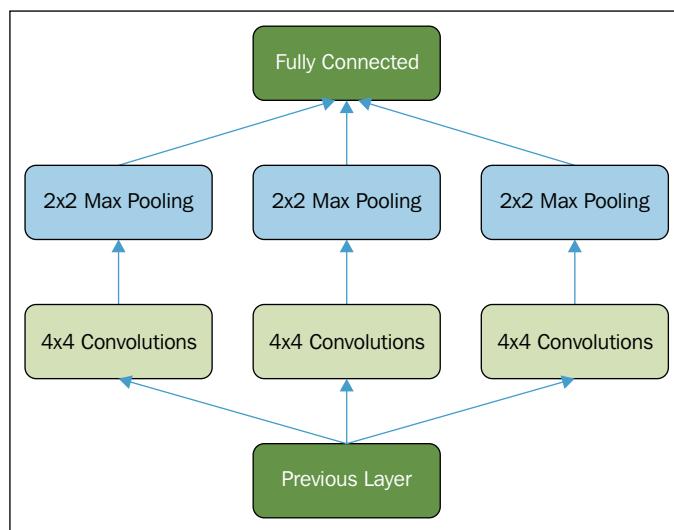
Each layer is partially-connected, as we discussed earlier, with the MLP being a fully connected layer. At each layer, multiple feature maps (channels) are employed; this gives us the advantage of being able to create more complex sets of filters. As we'll see, using multiple channels within a layer is a powerful technique employed in advanced use cases.

It's common to use max-pooling layers to reduce the dimensionality of the output to match the input as well as generally manage output volumes. How pooling is implemented, particularly in regard to the relative position of convolutional and pooling layers, is an element that tends to vary between implementations. It's generally common to develop a layer as a set of operations that feed into, and are fed into, a single **Fully Connected** layer, as shown in the following example:

While this network structure wouldn't work in practice, it's a helpful illustration of the fact that a network can be constructed from the components you've learned about in a number of ways. How this network is structured and how complex it becomes should be motivated by the challenge the network is intended to solve. Different problems can call for very different solutions.

In the case of the LeNet implementation that we'll be working with later in this chapter, each layer contains multiple convolutional layers in parallel with a max-pooling layer following each. Diagrammatically, a LeNet layer looks like the following image:
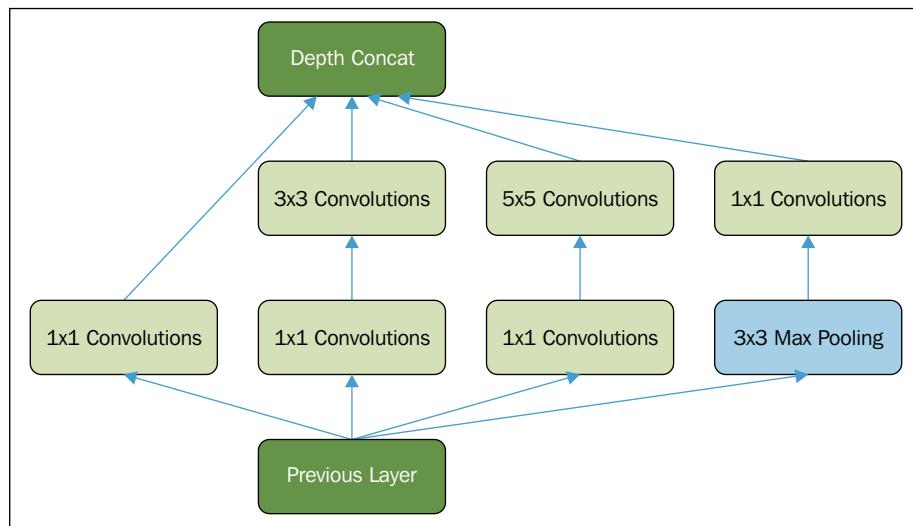
This architecture will enable us to start looking at some initial use cases quickly and easily, but in general won't perform well for some of the state-of-the-art applications we'll run into later in this book. Given this fact, there are some more extensive deep learning architectures designed to tackle the most challenging problems, whose topologies are worth discussing. One of the best-known convnet architectures is Google's **Inception** network, now more commonly known as GoogLeNet.

GoogLeNet was designed to tackle computer vision challenges involving Internet-quality image data, that is, images that have been captured in real contexts where the pose, lighting, occlusion, and clutter of images vary significantly. GoogLeNet was applied to the 2014 ImageNet challenge with noteworthy success, achieving only 6.7% error rate on the test dataset. ImageNet images are small, high-granularity images taken from many, varied classes. Multiple classes may appear very similar (such as varieties of tree) and the network architecture must be able to find increasingly challenging class distinctions to succeed. For a concrete example, consider the following ImageNet image:

Given the demands of this problem, the GoogLeNet architecture used to win ImageNet 14 departs from the LeNet model in several key ways. GoogLeNet's basic layer design is known as the Inception module and is made up of the following components:

The 1 x 1 convolutional layers used here are followed by **Rectified Linear Units** (**ReLU**). This approach is heavily used in speech and audio modeling contexts as ReLU can be used to effectively train deep models without pretraining and without facing some of the gradient vanishing problems that challenge other activation types. More information on ReLU is provided in the *Further reading* section of this chapter. The **DepthConcat** element provides a concatenation function, which consolidates the outputs of multiple units and substantially improves training time.

GoogLeNet chains layers of this type to create a full network. Indeed, the repetition of inception modules through GoogLeNet (nine times!) suggests that **Network In Network** (**NIN**) (deep architectures created from chained network modules) approaches are going to continue to be a serious contender in deep learning circles. The paper describing GoogLeNet and demonstrating how inception models were integrated into the network is provided in the Further reading section of this chapter.

Beyond the regularity of Inception module stacking, GoogLeNet has a few further surprises to throw at us. The first few layers are typically more straightforward with single-channel convolutional and max-pooling layers used at first. Additionally, at several points, GoogLeNet introduced a branch off the main structure using an average-pool layer, feeding into auxiliary softmax classifiers. The purpose of these classifiers was to improve the gradient signal that gets propagated back in lower layers of the network, enabling stronger performance at the early and middle network layers. Instead of one huge and potentially vague backpropagation process stemming from the final layer of the network, GoogLeNet instead has several intermediary update sources.

What's really important to take from this implementation is that GoogLeNet and other top convnet architectures are mainly successful because they are able to find effective configurations using the highly available components that we've discussed in this chapter. Now that we've had a chance to discuss the architecture and components of a convolutional net and the opportunity to discuss how these components are used to construct some highly advanced networks, it's time to apply the techniques to solve a problem of our own!

Applying a CNN
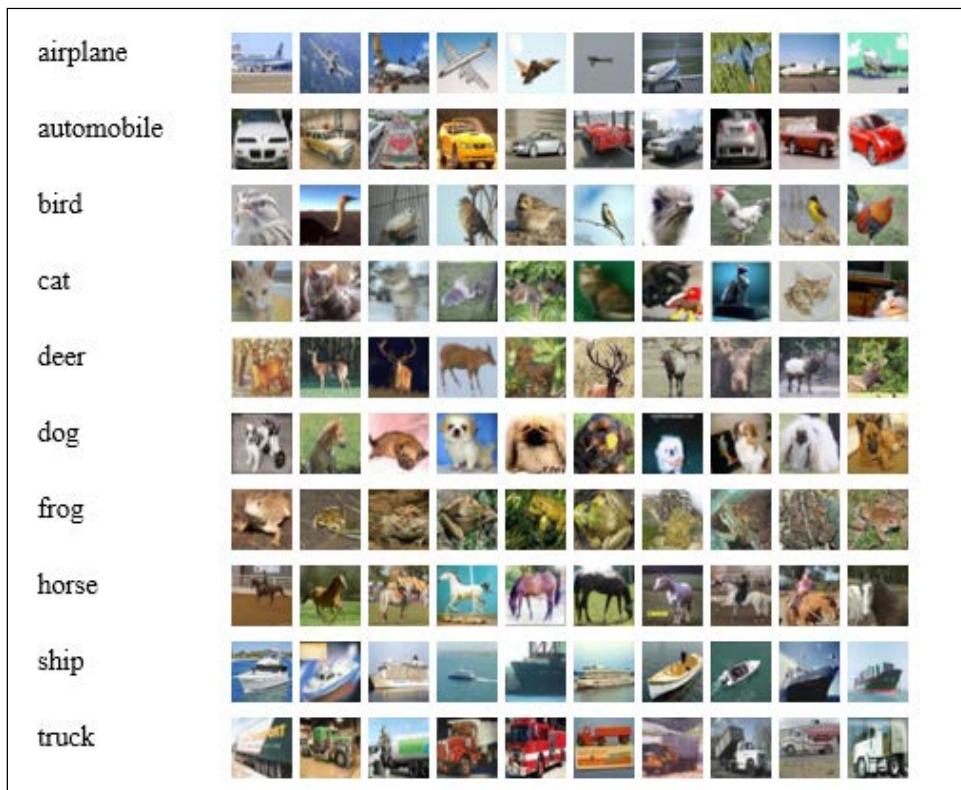
We'll be working with image data to try out our convnet. The image data that we worked with in earlier chapters, including the MNIST digit dataset, was a useful training dataset (with many valuable real-world applications such as automated check reading!). However, it differs from almost all photographic or video data in an important way; most visual data is highly noisy.

Problem variables can include pose, lighting, occlusion, and clutter, which may be expressed independently or in conjunction in huge variety. This means that the task of creating a function that is invariant to all properties of noise in the dataset is challenging; the function is typically very complex and nonlinear. In *Chapter 7, Feature Engineering Part II*, we'll discuss how techniques such as whitening can help mitigate some of these challenges, but as we'll see, even such techniques by themselves are insufficient to yield good classification (at least, without a very large investment of time!). By far, the most efficient solution to the problem of noise in image data, as we've already seen in multiple contexts, is to use a deep architecture rather than a broad one (that is, a neural network with few, high-dimensional layers, which is vulnerable to problematic overfitting and generalizability problems).

From discussions in previous chapters, the reasons for a deep architecture may already be clear; successive layers of a deep architecture reuse the reasoning and computation performed in preceding layers. Deep architectures can thus build a representation that is sequentially improved by successive layers of the network without performing extensive recalculation on any individual layer. This makes the challenging task of classifying large datasets of noisy photograph data achievable to a high level of accuracy in a relatively short time, without extensive feature engineering.

Now that we've discussed the challenges of modeling image data and advantages of a deep architecture in such contexts, let's apply a convnet to a real-world classification problem.

As in preceding chapters, we're going to start out with a toy example, which we'll use to familiarize ourselves with the architecture of our deep network. This time, we're going to take on a classic image processing challenge, CIFAR-10. CIFAR-10 is a dataset of 60,000 32 x 32 color images in 10 classes, with each class containing 6,000 images. The data is already split into five training batches, with one test batch. The classes and some images from each dataset are as follows:

While the industry has—to an extent—moved on to tackle other datasets such as ImageNet, CIFAR-10 was long regarded as the bar to reach in terms of image classification, with a great many data scientists attempting to create architectures that classify the dataset to human levels of accuracy, where human error rate is estimated at around 6%.

In November 2014, Kaggle ran a contest whose objective was to classify CIFAR-10 as accurately as possible. This contest's highest-scoring entry produced 95.55% classification accuracy, with the result using convolutional networks and a Network-in-Network approach. We'll discuss the challenge of classifying this dataset, as well as some of the more advanced techniques we can bring to bear, in *Chapter 8*, *Ensemble Methods*; for now, let's begin by having a go at classification with a convolutional network.

For our first attempt, we'll apply a fairly simple convolutional network with the following objectives:

- Applying a filter to the image and view the output
- Seeing the weights that our convnet created
- Understanding the difference between the outputs of effective and ineffective networks

In this chapter, we're going to take an approach that we haven't taken before, which will be of huge importance to you when you come to use these techniques in the wild. We saw earlier in this chapter how the deep architectures developed to solve different problems may differ structurally in many ways.

It's important to be able to create problem-specific network architectures so that we can adapt our implementation to fit a range of real-world problems. To do this, we'll be constructing our network using components that are modular and can be recombined in almost any way necessary, without too much additional effort. We saw the impact of modularity earlier in this chapter, and it's worth exploring how to apply this effect to our own networks.

As we discussed earlier in the chapter, convnets become particularly powerful when tasked to classify very large and varied datasets of up to tens or hundreds of thousands of images. As such, let's be a little ambitious and see whether we can apply a convnet to classify CIFAR-10.

In setting up our convolutional network, we'll begin by defining a useable class and initializing the relevant network parameters, particularly weights and biases. This approach will be familiar to readers of the preceding chapters.

```
class LeNetConvPoolLayer(object):

    def __init__(self, rng, input, filter_shape, image_shape,
    poolsize=(2, 2)):

        assert image_shape[1] == filter_shape[1]
        self.input = input

        fan_in = numpy.prod(filter_shape[1:])
        fan_out = (filter_shape[0] * numpy.prod(filter_shape[2:])
                    numpy.prod(poolsize))

        W_bound = numpy.sqrt(6. / (fan_in + fan_out))
        self.W = theano.shared(
            numpy.asarray(
                rng.uniform(low=-W_bound, high=W_bound,
                size=filter_shape),
                dtype=theano.config.floatX
            ),
            borrow=True
        )
```

Before moving on to create the biases, it's worth reviewing what we have thus far. The `LeNetConvPoolLayer` class is intended to implement one full convolutional and pooling layer as per the LeNet layer structure. This class contains several useful initial parameters.

From previous chapters, we're familiar with the `rng` parameter used to initialize weights to random values. We can also recognize the `input` parameter. As in most cases, image input tends to take the form of a symbolic image tensor. This image input is shaped by the `image_shape` parameter; this is a tuple or list of length 4 describing the dimensions of the input. As we move through successive layers, `image_shape` will reduce increasingly. As a tuple, the dimensions of `image_shape` simply specify the height and width of the input. As a list of length 4, the parameters, in order, are as follows:

- The batch size
- The number of input feature maps
- The height of the input image
- The width of the input image

While `image_shape` specifies the size of the input, `filter_shape` specifies the dimensions of the filter. As a list of length 4, the parameters, in order, are as follows:

- The number of filters (channels) to be applied
- The number of input feature maps
- The height of the filter
- The width of the filter

However, the height and width may be entered without any additional parameters. The final parameter here, `poolsize`, describes the downsizing factor. This is expressed as a list of length 2, the first element being the number of rows and the second — the number of columns.

Having defined these values, we immediately apply them to define the `LeNetConvPoolLayer` class better. In defining `fan_in`, we set the inputs to each hidden unit to be a multiple of the number of input feature maps — the filter height and width. Simply enough, we also define `fan_out`, a gradient that's calculated as a multiple of the number of output feature maps — the feature height and width — divided by the pooling size.

Next, we move on to defining the bias as a set of one-dimensional tensors, one for each output feature map:

```
b_values = numpy.zeros((filter_shape[0],),
dtype=theano.config.floatX)
self.b = theano.shared(value=b_values, borrow=True)

conv_out = conv.conv2d(
    input=input,
    filters=self.W,
    filter_shape=filter_shape,
    image_shape=image_shape
)
```

With this single function call, we've defined a convolution operation that uses the filters we previously defined. At times, it can be a little staggering to see how much theory needs to be known to effectively apply a single function! The next step is to create a similar pooling operation using `max_pool_2d`:

```
pooled_out = downsample.max_pool_2d(
    input=conv_out,
    ds=poolsize,
```

```
        ignore_border=True
    )


    self.output = T.tanh(pooled_out + self.b.dimshuffle('x',
                0, 'x', 'x'))

    self.params = [self.W, self.b]

    self.input = input
```

Finally, we add the bias term, first reshaping it to be a tensor of shape (`1`, `n_filters`, `1`, `1`). This has the simple effect of causing the bias to affect every feature map and minibatch. At this point, we have all of the components we need to build a basic convnet. Let's move on to create our own network:

```
x = T.matrix('x')
y = T.ivector('y')
```

This process is fairly simple. We build the layers in order, passing parameters to the class that we previously specified. Let's start by building our first layer:

```
layer0_input = x.reshape((batch_size, 1, 32, 32))

layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 1, 32, 32),
    filter_shape=(nkerns[0], 1, 5, 5),
    poolsize=(2, 2)
)
```

We begin by reshaping the input to spread it across all of the intended minibatches. As the CIFAR-10 images are of a 32 x 32 dimension, we've used this input size for the height and width dimensions. The filtering process reduces the size of this input to 32- 5+1 in each dimension, or 28. Pooling reduces this by half in each dimension to create an output layer of shape (`batch_size`, `nkerns[0]`, `14`, `14`).

This is a completed first layer. Next, we can attach a second layer to this using the same code:

```
layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 14, 14),
    filter_shape=(nkerns[1], nkerns[0], 5, 5),
    poolsize=(2, 2)
)
```

As per the previous layer, the output shape for this layer is (`batch_size, nkerns[1], 5, 5`). So far, so good! Let's feed this output to the next, fully-connected sigmoid layer. To begin with, we need to flatten the input shape to two dimensions. With the values that we've fed to the network so far, the input will be a matrix of shape *(500, 1250)*. As such, we'll set up an appropriate `layer2`:

```
layer2_input = layer1.output.flatten(2)

layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 5 * 5
    n_out=500,
    activation=T.tanh
)
```

This leaves us in a good place to finish this network's architecture, by adding a final, logistic regression layer that calculates the values of the fully-connected sigmoid layer.

Let's try out this code:

```
x = T.matrix(CIFAR-10_train)
y = T.ivector(CIFAR-10_test)
```

Chapter_4/convolutional_mlp.py

The results that we obtained were as follows:

Optimization complete.

Best validation score of 0.885725 % obtained at iteration 17400, with test performance 0.902508 %

The code for file convolutional_mlp.py ran for 26.50m

This accuracy score, at validation, is reasonably good. It's not at a human level of accuracy, which, as we established, is roughly 94%. Equally, it is not the best score that we could achieve with a convnet.

For instance, the Further Reading section of this chapter refers to a convnet implemented in Torch using a combination of dropout (which we studied in *Chapter 3*, *Stacked Denoising Autoencoders*) and **Batch Normalization** (a normalization technique intended to reduce covariate drift during the training process; refer to the Further Reading section for further technical notes and papers on this technique), which scored 92.45% validation accuracy.

A score of 88.57% is, however, in the same ballpark and can give us confidence that we're within striking distance of an effective network architecture for the CIFAR-10 problem. More importantly, you've learned a lot about how to configure and train a convolutional neural network effectively.

Further Reading

The glut of recent interest in Convolutional Networks means that we're spoiled for choice for further reading. One good option for an unfamiliar reader is the course notes from Andrej Karpathy's course: `http://cs231n.github.io/convolutional-networks/`.

For readers with an interest in the deeper details of specific best-in-class implementations, some of the networks referenced in this chapter were the following:

Google's GoogLeNet (`http://www.cs.unc.edu/~wliu/papers/GoogLeNet.pdf`)

Google Deepmind's Go-playing program AlphaGo (`https://gogameguru.com/i/2016/03/deepmind-mastering-go.pdf`)

Facebook's DeepFace architecture for facial recognition (`https://www.cs.toronto.edu/~ranzato/publications/taigman_cvpr14.pdf`)

The ImageNet LSVRC-2010 contest winning network, described here by Krizhevsky, Sutskever and Hinton (`http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf`)

Finally, Sergey Zagoruyko's Torch implementation of a ConvNet with Batch normalization is available here: `http://torch.ch/blog/2015/07/30/cifar.html`.

Summary

In this chapter, we covered a lot of ground. We began by introducing a new kind of neural network, the convnet. We explored the theory and architecture of a convnet in the most ubiquitous form and also by discussing some state-of the-art network design principles that have been developing as recently as mid-2015 in organizations such as Google and Baidu. We built an understanding of the topology and also of how the network operates.

Following this, we began to work with the convnet itself, applying it to the CIFAR-10 dataset. We used modular convnet code to create a functional architecture that reached a reasonable level of accuracy in classifying 10-class image data. While we're definitely still at some distance from human levels of accuracy, we're gradually closing the gap! *Chapter 8*, *Ensemble Methods* will pick up from what you learned here, taking these techniques and their application to the next level.

5

Semi-Supervised Learning

Introduction

In previous chapters, we've tackled a range of data challenges using advanced techniques. In each case, we've applied our techniques to datasets with reasonable success.

In many regards, though, we've had it pretty easy. Our data has been largely derived from canonical and well-prepared sources so we haven't had to do a great deal of preparation. In the real world, though, there are few datasets like this (except, perhaps, the ones that we're able to specify ourselves!). In particular, it is rare and improbable to come across a dataset in the wild, which has class labels available. Without labels on a sufficient portion of the dataset, we find ourselves unable to build a classifier that can accurately predict labels on validation or test data. So, what do we do?

The common solution is attempt to tag our data manually; not only is this time-consuming, but it also suffers from certain types of human error (which are especially common with high-dimensional datasets, where a human observer is unable to identify class boundaries as well as a computational approach might).

A fairly new and quite exciting alternative approach is to use **semi-supervised learning** to apply labels to unlabeled data via capturing the shape of underlying distributions. Semi-supervised learning has been growing in popularity over the last decade for its ability to save large amounts of annotation time, where annotation, if possible, may potentially require human expertise or specialist equipment. Contexts where this has proven to be particularly valuable have been natural language parsing and speech signal analysis; in both areas, manual annotation has proven to be complex and time-consuming.

In this chapter, you're going to learn how to apply several semi-supervised learning techniques, including, **Contrastive Pessimistic Likelihood Estimation** (**CPLE**), self learning, and S3VM. These techniques will enable us to label training data in a range of otherwise problematic contexts. You'll learn to identify the capabilities and limitations of semi-supervised techniques. We'll use a number of recent Python libraries developed on top of scikit-learn to apply semi-supervised techniques to several use cases, including audio signal data.

Let's get started!

Understanding semi-supervised learning

The most persistent cost in performing machine learning is the creation of tagged data for training purposes. Datasets tend not to come with class labels provided due to the circularity of the situation; one needs a trained classification technique to generate class labels, but cannot train the technique without labeled training and test data. As mentioned, tagging data manually or via test processes is one option, but this can be prohibitively time-consuming, costly (particularly for medical tests), challenging to organize, and prone to error (with large or complex datasets). Semi-supervised techniques suggest a better way to break this deadlock.

Semi-supervised learning techniques use both unlabeled and labeled data to create better learning techniques than can be created with either unlabeled or labeled data individually. There is a family of techniques that exists in a space between supervised (with labeled data) and unsupervised (with unlabeled data) learning.

The main types of technique that exist in this group are semi-supervised techniques, transductive techniques, and active learning techniques, as well as a broad set of other methods.

Semi-supervised techniques leave a set of test data out of the training process so as to perform testing at a later stage. Transductive techniques, meanwhile, are purely intended to develop labels for unlabeled data. There may not be a test process embedded in a transductive technique and there may not be labeled data available for use.

In this chapter, we'll focus on a set of semi-supervised techniques that deliver powerful dataset labeling capability in very familiar formats. A lot of the techniques that we'll be discussing are useable as wrappers around familiar, pre-existing classifiers, from linear regression classifiers to SVMs. As such, many of them can be run using estimators from Scikit-learn. We'll begin by applying a linear regression classifier to test cases before moving on to apply an SVM with semi-supervised extensions.

Semi-supervised algorithms in action

We've discussed what semi-supervised learning is, why we want to engage in it, and what some of the general realities of employing semi-supervised algorithms are. We've gone about as far as we can with general descriptions. Over the next few pages, we'll move from this general understanding to develop an ability to use a semi-supervised application effectively.

Self-training

Self-training is the simplest semi-supervised learning method and can also be the fastest. Self-training algorithms see an application in multiple contexts, including NLP and computer vision; as we'll see, they can present both substantial value and significant risks.
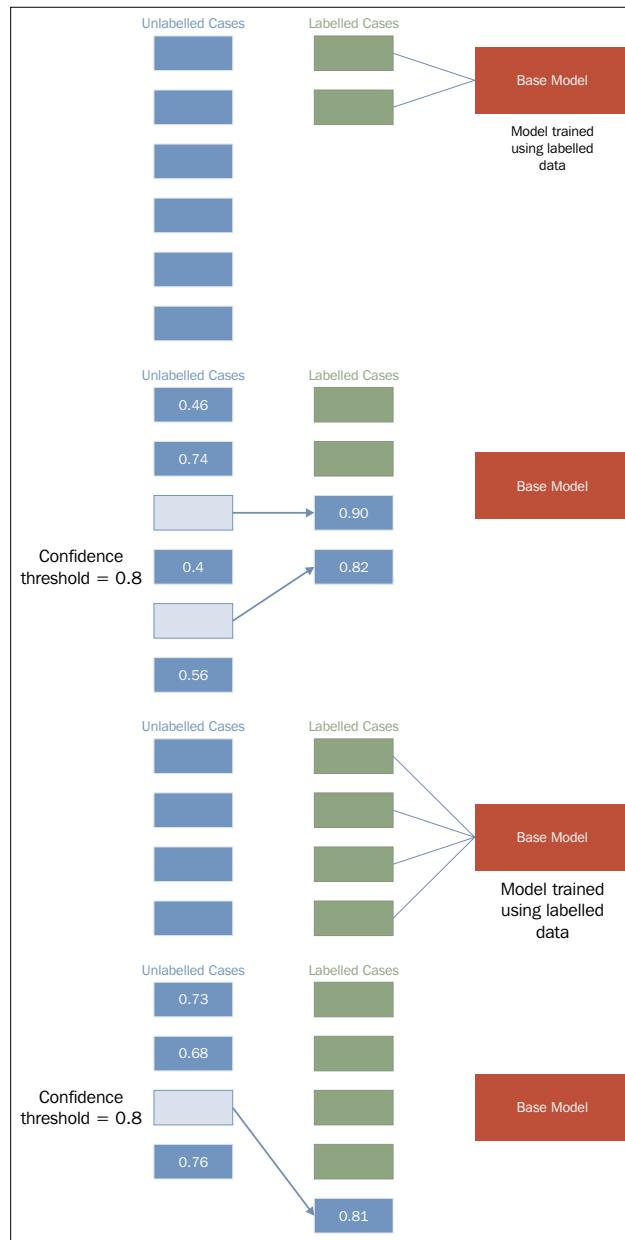
The objective of self-training is to combine information from unlabeled cases with that of labeled cases to iteratively identify labels for the dataset's unlabeled examples. On each iteration, the labeled training set is enlarged until the entire dataset is labeled.

The self-training algorithm is typically applied as a wrapper to a base model. In this chapter, we'll be using an SVM as the base for our self-training model. The self-training algorithm is quite simple and contains very few steps, as follows:

1. A set of labeled data is used to predict labels for a set of unlabeled data. (This may be all unlabeled data or part of it.)
2. Confidence is calculated for all newly labeled cases.
3. Cases are selected from the newly labeled data to be kept for the next iteration.

4. The model trains on all labeled cases, including cases selected in previous iterations.

5. The model iterates through steps 1 to 4 until it successfully converges.

Presented graphically, this process looks as follows:

Upon completing training, the self-trained model would be tested and validated. This may be done via cross-validation or even using held-out, labeled data, should this exist.

Self-training provides real power and time saving, but is also a risky process. In order to understand what to look out for and how to apply self-training to your own classification algorithms, let's look in more detail at how the algorithm works.

To support this discussion, we're going to work with code from the semisup-learn GitHub repository. In order to use this code, we'll need to clone the relevant GitHub repository. Instructions for this are located in *Appendix A*.

Implementing self-training

The first step in each iteration of self-training is one in which class labels are generated for unlabeled cases. This is achieved by first creating a `SelfLearningModel` class, which takes a base supervised model (`basemodel`) and an iteration limit as arguments. As we'll see later in this chapter, an iteration limit can be explicitly specified or provided as a function of classification accuracy (that is, convergence). The `prob_threshold` parameter provides a minimum quality bar for label acceptance; any projected label that scores at less than this level will be rejected. Again, we'll see in later examples that there are alternatives to providing a hardcoded threshold value.

```
class SelfLearningModel(BaseEstimator):


def __init__(self, basemodel, max_iter = 200, prob_threshold = 0.8):
   self.model = basemodel
   self.max_iter = max_iter
   self.prob_threshold = prob_threshold
```

Having defined the shell of the `SelfLearningModel` class, the next step is to define functions for the process of semi-supervised model fitting:

```
def fit(self, X, y):
   unlabeledX = X[y==-1, :]
   labeledX = X[y!=-1, :]
   labeledy = y[y!=-1]

   self.model.fit(labeledX, labeledy)
   unlabeledy = self.predict(unlabeledX)
   unlabeledprob = self.predict_proba(unlabeledX)
   unlabeledy_old = []

   i = 0
```

The X parameter is a matrix of input data, whose shape is equivalent to `[n_samples, n_features]`. X is used to create a matrix of `[n_samples, n_samples]` size. The y parameter, meanwhile, is an array of labels. Unlabeled points are marked as `-1` in y. From X, the `unlabeledX` and `labeledX` parameters are created quite simply by operations over X that select elements in X whose position corresponds to a `-1` label in y. The `labeledy` parameter performs a similar selection over y. (Naturally, we're not that interested in the unlabeled samples of y as a variable, but we need the labels that do exist for classification attempts!)

The actual process of label prediction is achieved, first, using sklearn's predict operation. The `unlabeledy` parameter is generated using sklearn's `predict` method, while the `predict_proba` method is used to calculate probabilities for each projected label. These probabilities are stored in `unlabeledprob`.

Scikit-learn's `predict` and `predict_proba` methods work to `predict` class labels and the probability of class labeling being correct, respectively. As we'll be applying both of these methods within several of our semi-supervised algorithms, it's informative to understand how they actually work.

The `predict` method produces class predictions for input data. It does so via a set of binary classifiers (that is, classifiers that attempt to differentiate only two classes). A full model with n-many classes contains a set of binary classifiers as follows:

$$\frac{n*(n-1)}{2}$$

In order to make a prediction for a given case, all classifiers whose scores exceed zero, vote for a class label to apply to that case. The class with the most votes (and not, say, the highest sum classifier score) is identified. This is referred to as a one-versus-one prediction method and is a fairly common approach.

Meanwhile, `predict_proba` works by invoking **Platt calibration**, a technique that allows the outputs of a classification model to be transformed into a probability distribution over the classes. This involves first training the base model in question, fitting a regression model to the classifier's scores:

$$P(y \mid X) = \frac{1}{\left(1 + \exp\left(A * f(X) + B\right)\right)}$$

This model can then be optimized (through scalar parameters *A* and *B*) using a maximum likelihood method. In the case of our self-training model, `predict_proba` allows us to fit a regression model to the classifier's scores and thus calculate probabilities for each class label. This is extremely helpful!

Next, we need a loop for iteration. The following code describes a `while` loop that executes until there are no cases left in `unlabeledy_old` (a copy of `unlabeledy`) or until the max iteration count is reached. On each iteration, a labeling attempt is made for each case that does not have a label whose probability exceeds the probability threshold (`prob_threshold`):

```
while (len(unlabeledy_old) == 0 or
    numpy.any(unlabeledy!=unlabeledy_old)) and i < self.max_iter:
    unlabeledy_old = numpy.copy(unlabeledy)
    uidx = numpy.where((unlabeledprob[:, 0] > self.prob_threshold)
    | (unlabeledprob[:, 1] > self.prob_threshold))[0]
```

The `self.model.fit` method then attempts to fit a model to the unlabeled data. This unlabeled data is presented in a matrix of size `[n_samples, n_samples]` (as referred to earlier in this chapter). This matrix is created by appending (with `vstack` and `hstack`) the unlabeled cases:

```
self.model.fit(numpy.vstack((labeledX, unlabeledX[uidx, :])),
    numpy.hstack((labeledy, unlabeledy_old[uidx])))
```

Finally, the iteration performs label predictions, followed by probability predictions for those labels.

```
unlabeledy = self.predict(unlabeledX)
unlabeledprob = self.predict_proba(unlabeledX)
i += 1
```

On the next iteration, the model will perform the same process, this time taking the newly labeled data whose probability predictions exceeded the threshold as part of the dataset used in the `model.fit` step.

If one's model does not already include a classification method that can generate label predictions (like the `predict_proba` method available in sklearn's SVM implementation), it is possible to introduce one. The following code checks for the `predict_proba` method and introduces `Platt scaling` of generated labels if this method is not found:

```
if not getattr(self.model, "predict_proba", None):
    self.plattlr = LR()
    preds = self.model.predict(labeledX)
    self.plattlr.fit( preds.reshape( -1, 1 ), labeledy )

return self



def predict_proba(self, X):
        if getattr(self.model, "predict_proba", None):
        return self.model.predict_proba(X)
        else:
            preds = self.model.predict(X)
            return self.plattlr.predict_proba(preds.reshape( -1, 1 ))
```

Once we have this much in place, we can begin applying our self-training architecture. To do so, let's grab a dataset and start working!

For this example, we'll use a simple linear regression classifier, with **Stochastic Gradient Descent** (**SGD**) as our learning component as our base model (`basemodel`). The input dataset will be the statlog `heart` dataset, obtained from `www.mldata.org`. This dataset is provided in the GitHub repository accompanying this chapter.

The `heart` dataset is a two-class dataset, where the classes are the absence or presence of a heart disease. There are no missing values across the 270 cases for any of its 13 features. This data is unlabeled and many of the variables needed are usually captured via expensive and sometimes inconvenient tests. The variables are as follows:

- `age`

- `sex`

- `chest pain type (4 values)`

- `resting blood pressure`

- `serum cholestoral in mg/dl`

- `fasting blood sugar > 120 mg/dl`

- `resting electrocardiographic results (values 0,1,2)`

- `maximum heart rate achieved`

- `exercise induced angina`

- `10. oldpeak = ST depression induced by exercise relative to rest`

- `the slope of the peak exercise ST segment`

- `number of major vessels (0-3) colored by flourosopy`

- `thal: 3 = normal; 6 = fixed defect; 7 = reversable defect`

Lets get started with the `Heart` dataset by loading in the data, then fitting a model to it:

```
heart = fetch_mldata("heart")
X = heart.data
ytrue = np.copy(heart.target)
ytrue[ytrue==-1]=0

labeled_N = 2
ys = np.array([-1]*len(ytrue)) # -1 denotes unlabeled point
random_labeled_points = random.sample(np.where(ytrue == 0)[0],
labeled_N/2)+\random.sample(np.where(ytrue == 1)[0], labeled_N/2)
ys[random_labeled_points] = ytrue[random_labeled_points]

basemodel = SGDClassifier(loss='log', penalty='l1')

basemodel.fit(X[random_labeled_points, :], ys[random_labeled_points])
print "supervised log.reg. score", basemodel.score(X, ytrue)

ssmodel = SelfLearningModel(basemodel)
ssmodel.fit(X, ys)
print "self-learning log.reg. score", ssmodel.score(X, ytrue)
```
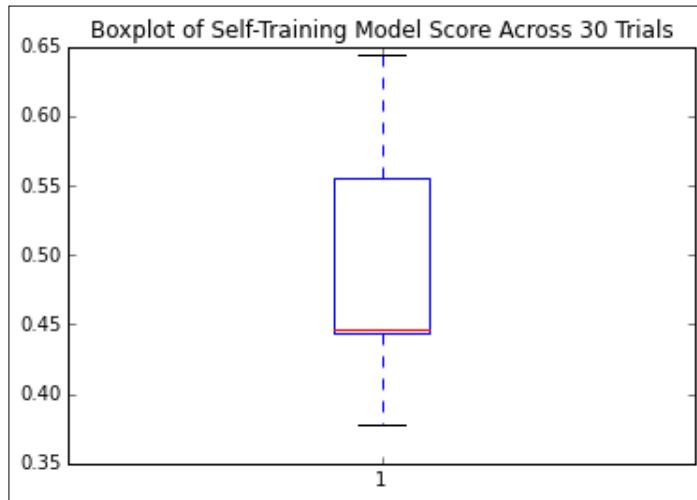
Attempting this yields moderate, but not excellent, results:

```
self-learning log.reg. score 0.470347
```

However, over 1,000 trials, we find that the quality of our outputs is quite variant:

Given that we're looking at classification accuracy scores for sets of real-world and unlabeled data, this isn't a terrible result, but I don't think we should be satisfied with it. We're still labeling more than half of our cases incorrectly!

We need to understand the problem a little better; right now, it isn't clear what's going wrong or how we can improve on our results. Let's figure this out by returning to the theory around self-training to understand how we can diagnose and improve our implementation.

Finessing your self-training implementation

In the previous section, we discussed the creation of self-training algorithms and tried out an implementation. However, what we saw during our first trial was that our results, while demonstrating the potential of self-training, left room for growth. Both the accuracy and variance of our results were questionable.

Self-training can be a fragile process. If an element of the algorithm is ill-configured or the input data contains peculiarities, it is very likely that the iterative process will fail once and continue to compound that error by reintroducing incorrectly labeled data to future labeling steps. As the self-training algorithm iteratively feeds itself, garbage in, garbage out is a very real concern.

There are several quite common flavors of risk that should be called out. In some cases, labeled data may not add more useful information. This is particularly common in the first few iterations, and understandably so! In general, unlabeled cases that are most easily labeled are the ones that are most similar to existing labeled cases. However, while it's easy to generate high-probability labels for these cases, there's no guarantee that their addition to the labeled set will make it easier to label during subsequent iterations.

Unfortunately, this can sometimes lead to a situation in which cases are being added that have no real effect on classification while classification accuracy in general deteriorates. Even worse, adding cases that are similar to pre-existing cases in enough respects to make them easy to label, but that actually misguide the classifier's decision boundary, can introduce misclassification increases.

Diagnosing what went wrong with a self-training model can sometimes be difficult, but as always, a few well-chosen plots add a lot of clarity to the situation. As this type of error occurs particularly often within the first few iterations, simply adding an element to the label prediction loop that writes the current classification accuracy allows us to understand how accuracy trended during early iterations.

Once the issue has been identified, there are a few possible solutions. If enough labeled data exists, a simple solution is to attempt to use a more diverse set of labeled data to kick-start the process.

While the impulse might be to use all of the labeled data, we'll see later in this chapter that self-training models are vulnerable to overfitting—a risk that forces us to hold on to some data for validation purposes. A promising option is to use multiple subsets of our dataset to train multiple self-training model instances. Doing so, particularly over several trials, can help us understand the impact of our input data on our self-training models performance.

In *Chapter 8*, *Ensemble Methods*, we'll explore some options around ensembles that will enable us to use multiple self-training models together to yield predictions. When ensembling is accessible to us, we can even consider applying multiple sampling techniques in parallel.

If we don't want to solve this problem with quantity, though, perhaps we can solve it by improving quality. One solution is to create an appropriately diverse subset of the labeled data through selection. There isn't a hard limit on the number of labeled cases that works well as a minimum amount to start up a self-training implementation. While you could hypothetically start working with even one labeled case per class (as we did in our preceding training example), it'll quickly become obvious that training against a more diverse and overlapping set of classes benefits from more labeled data.

Another class of error that a self-training model is particularly vulnerable to is biased selection. Our naïve assumption is that the selection of data during each iteration is, at worst, only slightly biased (favoring one class only slightly more than others). The reality is that this is not a safe assumption. There are several factors that can influence the likelihood of biased selection, with the most likely culprit being disproportionate sampling from one class.

If the dataset as a whole, or the labeled subsets used, are biased toward one class, then the risk increases that your self-training classifier will overfit. This only compounds the problem as the cases provided for the next iteration are liable to be insufficiently diverse to solve the problem; whatever incorrect decision boundary was set up by the self-training algorithm will be set where it is—overfit to a subset of the data. Numerical disparity between each class' count of cases is the main symptom here, but the more usual methods to spot overfitting can also be helpful in diagnosing problems around selection bias.

This reference to the usual methods of spotting overfitting is worth expanding on because techniques to identify overfitting are highly valuable! These techniques are typically referred to as validation techniques. The fundamental concept underpinning validation techniques is that one has two sets of data—one that is used to build a model, and the other is used to test it.

The most effective validation technique is independent validation, the simplest form of which involves waiting to determine whether predictions are accurate. This obviously isn't always (or even, often) possible!

Given that it may not be possible to perform independent validation, the best bet is to hold out a subset of your sample. This is referred to as sample splitting and is the foundation of modern validation techniques. Most machine learning implementations refer to training, test, and validation datasets; this is a case of multilayered validation in action.

A third and critical validation tool is resampling, where subsets of the data are iteratively used to repeatedly validate the dataset. In *Chapter 1, Unsupervised Machine Learning*, we saw the use of v-fold cross-validation; cross-validation techniques are perhaps the best examples of resampling in action.

Beyond applicable techniques, it's a good idea to be mindful of the needed sample size required for the effective modeling of your data. There are no universal principles here, but I always rather liked the following rule of thumb:

If m points are required to determine a univariate regression line with sufficient precision, then it will take at least mn observations and perhaps $n!mn$ observations to appropriately characterize and evaluate a regression model with n variables.

Note that there is some tension between the suggested solutions to this problem (resampling, sample splitting, and validation techniques including cross-validation) and the preceding one. Namely, overfitting requires a more restrained use of subsets of the labeled training data, while bad starts are less likely to occur using more training data. For each specific problem, depending on the complexity of the data under analysis, there will be an appropriate balance to strike. By monitoring for signs of either type of problem, the appropriate action (whether that is an increase or decrease in the amount of labeled data used simultaneously in an iteration) can be taken at the right time.

A further class of risk introduced by self-training is that the introduction of unlabeled data almost always introduces noise. If dealing with datasets where part or all of the unlabeled cases are highly noisy, the amount of noise introduced may be sufficient to degrade classification accuracy.

> The idea of using data complexity and noise measures to understand the degree of noise in one's dataset is not new. Fortunately for us, quite a lot of good estimators already exist that we can take advantage of.
>
> There are two main groups of relative complexity measures. Some attempt to measure the overlap of values of different classes, or separability; measures in this group attempt to describe the degree of ambiguity of each class relative to the other classes. One good measure for such cases is the maximum **Fisher's discriminant ratio**, though maximum individual feature efficiency is also effective.
>
> Alternatively (and sometimes more simply), one can use the error function of a linear classifier to understand how separable the dataset's classes are from one another. By attempting to train a simple linear classifier on your dataset and observing the training error, one can immediately get a good understanding as to how linearly separable the classes are. Furthermore, measures related to this classifier (such as the fraction of points in the class boundary or the ratio of average intra/inter class nearest neighbor distance) can also be extremely helpful.
>
> There are other data complexity measures that specifically measure the density or geometry of the dataset. One good example is the fraction of maximum covering spheres. Again, helpful measures can be accessed by applying a linear classifier and including the nonlinearity of that classifier.

Improving the selection process

The key to the self-training algorithm working correctly is the accurate calculation of confidence for each label projection. Confidence calculation is the key to successful self-training.

During our first explanation of self-training, we used some simplistic values for certain parameters, including a parameter closely tied to confidence calculation. In selecting our labeled cases, we used a fixed confidence level for comparison against predicted probabilities, where we could've adopted any one of several different strategies:

- Adding all of the projected labels to the set of labeled data
- Using a confidence threshold to select only the few most confident labels to the set
- Adding all the projected labels to the labeled dataset and weighing each label by confidence

All in all, we've seen that self-training implementations present quite a lot of risk. They're prone to a number of training failures and are also subject to overfitting. To make matters worse, as the amount of unlabeled data increases, the accuracy of a self-training classifier becomes increasingly at risk.

Our next step will be to look at a very different self-training implementation. While conceptually similar to the algorithm that we worked with earlier in this chapter, the next technique we'll be looking at operates under different assumptions to yield very different results.

Contrastive Pessimistic Likelihood Estimation

In our preceding discovery and application of self-training techniques, we found self-training to be a powerful technique with significant risks. Particularly, we found a need for multiple diagnostic tools and some quite restrictive dataset conditions. While we can work around these problems by subsetting, identifying optimal labeled data, and attentively tracking performance for some datasets, some of these actions continue to be impossible for the very data that self-training would bring the most benefit to—data where labeling requires expensive tests, be those medical or scientific, with specialist knowledge and equipment.

In some cases, we end up with some self-training classifiers that are outperformed by their supervised counterparts, which is a pretty terrible state of affairs. Even worse, while a supervised classifier with labeled data will tend to improve in accuracy with additional cases, semi-supervised classifier performance can degrade as the dataset size increases. What we need, then, is a less naïve approach to semi-supervised learning. Our goal should be to find an approach that harnesses the benefits of semi-supervised learning while maintaining performance at least comparable with that of the same classifier under a supervised approach.

A very recent (May 2015) approach to self-supervised learning, CPLE, provides a more general way to perform semi-supervised parameter estimation. CPLE provides a rather remarkable advantage: it produces label predictions that have been demonstrated to consistently outperform those created by equivalent semi-supervised classifiers or by supervised classifiers working from the labeled data! In other words, when performing a linear discriminant analysis, for instance, it is advised that you perform a CPLE-based, semi-supervised analysis instead of a supervised one, as you will always obtain at least equivalent performance.

This is a pretty big claim and it needs substantiating. Let's start by building an understanding of how CPLE works before moving on to demonstrate its superior performance in real cases.

CPLE uses the familiar measure of maximized log-likelihood for parameter optimization. This can be thought of as the success condition; the model we'll develop is intended to optimize the maximized log-likelihood of our model's parameters. It is the specific guarantees and assumptions that CPLE incorporates that make the technique effective.

In order to create a better semi-supervised learner—one that improves on it's supervised alternative—CPLE takes the supervised estimates into account explicitly, using the loss incurred between the semi-supervised and supervised models as a training performance measure:

CPLE calculates the relative improvement of any semi-supervised estimate over the supervised solution. Where the supervised solution outperforms the semi-supervised estimate, the loss function shows this and the model can train to adjust the semi-supervised model to reduce this loss. Where the semi-supervised solution outperforms the supervised solution, the model can learn from the semi-supervised model by adjusting model parameters.

However, while this sounds excellent so far, there is a flaw in the theory that has to be addressed. The fact that data labels don't exist for a semi-supervised solution means that the posterior distribution (that CPLE would use to calculate loss) is inaccessible. CPLE's solution to this is to be pessimistic. The CPLE algorithm takes the **Cartesian product** of all label/prediction combinations and then selects the posterior distribution that minimizes the gain in likelihood.

In real-world machine learning contexts, this is a very safe approach. It delivers the classification accuracy of a supervised approach with semi-supervised performance improvement derived via conservative assumptions. In real applications, these conservative assumptions enable high performance under testing. Even better, CPLE can deliver particular performance improvements on some of the most challenging unsupervised learning cases, where the labeled data is a poor representation of the unlabeled data (by virtue of poor sampling from one or more classes or just because of a shortage of unlabeled cases).

In order to understand how much more effective CPLE can be than semi-supervised or supervised approaches, let's apply the technique to a practical problem. We'll once again work with the semisup-learn library, a specialist Python library, focused on semi-supervised learning, which extends scikit-learn to provide CPLE across any scikit-learn-provided classifier. We begin with a CPLE class:

```
class CPLELearningModel(BaseEstimator):

    def __init__(self, basemodel, pessimistic=True, predict_from_
probabilities = False, use_sample_weighting = True, max_iter=3000,
verbose = 1):
        self.model = basemodel
        self.pessimistic = pessimistic
        self.predict_from_probabilities = predict_from_probabilities
        self.use_sample_weighting = use_sample_weighting
        self.max_iter = max_iter
        self.verbose = verbose
```

We're already familiar with the concept of `basemodel`. Earlier in this chapter, we employed S3VMs and semi-supervised LDE's. In this situation, we'll again use an LDE; the goal of this first assay will be to try and exceed the results obtained by the semi-supervised LDE from earlier in this chapter. In fact, we're going to blow those results out of the water!

Before we do so, however, let's review the other parameter options. The `pessimistic` argument gives us an opportunity to use a non-pessimistic (optimistic) model. Instead of following the `pessimistic` method of minimizing the loss between unlabeled and labeled discriminative likelihood, an optimistic model aims to maximize likelihood. This can yield better results (mostly during training), but is significantly more risky. Here, we'll be working with pessimistic models.

The `predict_from_probabilities` parameter enables optimization by allowing a prediction to be generated from the probabilities of multiple data points at once. If we set this as true, our CPLE will set the prediction as `1` if the probability we're using for prediction is greater than the mean, or `0` otherwise. The alternative is to use the base model probabilities, which is generally preferable for performance reasons, unless we'll be calling `predict` across a number of cases.

We also have the option to `use_sample_weighting`, otherwise known as **soft labels** (but most familiar to us as posterior probabilities). We would normally take this opportunity, as soft labels enable greater flexibility than hard labels and are generally preferred (unless the model only supports hard class labels).

The first few parameters provide a means of stopping CPLE training, either at maximum iterations or after log-likelihood stops improving (typically because of convergence). The `bestdl` provides the best discriminative likelihood value and corresponding soft labels; these values are updated on each training iteration:

```
self.it = 0
self.noimprovementsince = 0
self.maxnoimprovementsince = 3

self.buffersize = 200
self.lastdls = [0]*self.buffersize

self.bestdl = numpy.infty
self.bestlbls = []


self.id = str(unichr(numpy.random.randint(26)+97))+str(unichr(
numpy.random.randint(26)+97))
```

The `discriminative_likelihood` function calculates the likelihood (for discriminative models—that is, models that aim to maximize the probability of a target—$y = 1$, conditional on the input, X) of an input.

> In this case, it's worth drawing your attention to the distinction between generative and discriminative models. While this isn't a basic concept, it can be fundamental in understanding why many classifiers have the goals that they do.
>
> A classification model takes input data and attempts to classify cases, assigning each case a label. There is more than one way to do this.
>
> One approach is to take the cases and attempt to draw a decision boundary between them. Then we can take each new case as it appears and identify which side of the boundary it falls on. This is a **discriminative** learning approach.
>
> Another approach is to attempt to model the distribution of each class individually. Once a model has been generated, the algorithm can use Bayes' rule to calculate the posterior distribution on the labels given input data. This approach is **generative** and is a very powerful approach with significant weaknesses (most of which tie into the question of how well we can model our classes). Generative approaches include Gaussian discriminant models (yes, that is a slightly confusing name) and a broad range of Bayesian models. More information, including some excellent recommended reading, is provided in the *Further reading* section of this chapter.

In this case, the function will be used on each iteration to calculate the likelihood of the predicted labels:

```
    def discriminative_likelihood(self, model, labeledData, labeledy =
None, unlabeledData = None, unlabeledWeights = None, unlabeledlambda =
1, gradient=[], alpha = 0.01):
        unlabeledy = (unlabeledWeights[:, 0]<0.5)*1
        uweights = numpy.copy(unlabeledWeights[:, 0])

        uweights[unlabeledy==1] = 1-uweights[unlabeledy==1]

        weights = numpy.hstack((numpy.ones(len(labeledy)), uweights))
        labels = numpy.hstack((labeledy, unlabeledy))
```

Having defined this much of our CPLE, we also need to define the fitting process for our supervised model. This uses familiar components, namely, `model.fit` and `model.predict_proba`, for probability prediction:

```
        if self.use_sample_weighting:
            model.fit(numpy.vstack((labeledData, unlabeledData)),
labels, sample_weight=weights)
        else:
```

```
            model.fit(numpy.vstack((labeledData, unlabeledData)),
    labels)



        P = model.predict_proba(labeledData)
```

In order to perform pessimistic CPLE, we need to derive both the labeled and unlabeled discriminative log likelihood. In order, we then perform `predict_proba` on both the labeled and unlabeled data:

```
        try:

            labeledDL = -sklearn.metrics.log_loss(labeledy, P)
        except Exception, e:
            print e
            P = model.predict_proba(labeledData)



        unlabeledP = model.predict_proba(unlabeledData)



        try:
            eps = 1e-15
            unlabeledP = numpy.clip(unlabeledP, eps, 1 - eps)
            unlabeledDL = numpy.average((unlabeledWeights*numpy.
    vstack((1-unlabeledy, unlabeledy)).T*numpy.log(unlabeledP)).
    sum(axis=1))
        except Exception, e:
            print e
            unlabeledP = model.predict_proba(unlabeledData)
```

Once we're able to calculate the discriminative log likelihood for both the labeled and unlabeled classification attempts, we can set an objective via the `discriminative_likelihood_objective` function. The goal here is to use the pessimistic (or optimistic, by choice) methodology to calculate `dl` on each iteration until the model converges or the maximum iteration count is hit.

On each iteration, a t-test is performed to determine whether the likelihoods have changed. Likelihoods should continue to change on each iteration preconvergence. Sharp-eyed readers may have noticed earlier in the chapter that three consecutive t-tests showing no change will cause the iteration to stop (this is configurable via the `maxnoimprovementsince` parameter):

```
        if self.pessimistic:
            dl = unlabeledlambda * unlabeledDL - labeledDL
        else:
            dl = - unlabeledlambda * unlabeledDL - labeledDL

        return dl


    def discriminative_likelihood_objective(self, model, labeledData,
labeledy = None, unlabeledData = None, unlabeledWeights = None,
unlabeledlambda = 1, gradient=[], alpha = 0.01):
        if self.it == 0:
            self.lastdls = [0]*self.buffersize

        dl = self.discriminative_likelihood(model, labeledData,
labeledy, unlabeledData, unlabeledWeights, unlabeledlambda, gradient,
alpha)

        self.it += 1
        self.lastdls[numpy.mod(self.it, len(self.lastdls))] = dl

        if numpy.mod(self.it, self.buffersize) == 0: # or True:
            improvement = numpy.mean((self.lastdls[(len(self.
lastdls)/2):])) - numpy.mean((self.lastdls[:(len(self.lastdls)/2)]))

            _, prob = scipy.stats.ttest_ind(self.lastdls[(len(self.
lastdls)/2):], self.lastdls[:(len(self.lastdls)/2)])

            noimprovement = prob > 0.1 and numpy.mean(self.
lastdls[(len(self.lastdls)/2):]) < numpy.mean(self.lastdls[:(len(self.
lastdls)/2)])
            if noimprovement:
                self.noimprovementsince += 1
                if self.noimprovementsince >= self.
maxnoimprovementsince:

                    self.noimprovementsince = 0
                    raise Exception(" converged.")
            else:
                self.noimprovementsince = 0
```

On each iteration, the algorithm saves the best discriminative likelihood and the best weight set for use in the next iteration:

```
if dl < self.bestdl:
    self.bestdl = dl
    self.bestlbls = numpy.copy(unlabeledWeights[:, 0])


return dl
```

One more element worth discussing is how the soft labels are created. We've discussed these earlier in the chapter. This is how they look in code:

```
f = lambda softlabels, grad=[]: self.discriminative_
likelihood_objective(self.model, labeledX, labeledy=labeledy,
unlabeledData=unlabeledX, unlabeledWeights=numpy.vstack((softlabels,
1-softlabels)).T, gradient=grad)

lblinit = numpy.random.random(len(unlabeledy))
```

In a nutshell, `softlabels` provide a probabilistic version of the discriminative likelihood calculation. In other words, they act as weights rather than hard, binary class labels. Soft labels are calculable using the `optimize` method:

```
try:
    self.it = 0
    opt = nlopt.opt(nlopt.GN_DIRECT_L_RAND, M)
    opt.set_lower_bounds(numpy.zeros(M))
    opt.set_upper_bounds(numpy.ones(M))
    opt.set_min_objective(f)
    opt.set_maxeval(self.max_iter)
    self.bestsoftlbl = opt.optimize(lblinit)
    print " max_iter exceeded."
except Exception, e:
    print e
    self.bestsoftlbl = self.bestlbls

if numpy.any(self.bestsoftlbl != self.bestlbls):
    self.bestsoftlbl = self.bestlbls
ll = f(self.bestsoftlbl)

unlabeledy = (self.bestsoftlbl<0.5)*1
uweights = numpy.copy(self.bestsoftlbl)

uweights[unlabeledy==1] = 1-uweights[unlabeledy==1]

weights = numpy.hstack((numpy.ones(len(labeledy)), uweights))
labels = numpy.hstack((labeledy, unlabeledy))
```

> For interested readers, optimize uses the **Newton conjugate gradient** method of calculating gradient descent to find optimal weight values. A reference to Newton conjugate gradient is provided in the Further reading section at the end of this chapter.

Once we understand how this works, the rest of the calculation is a straightforward comparison of the best supervised labels and soft labels, setting the `bestsoftlabel` parameter as the best label set. Following this, the discriminative likelihood is computed against the best label set and a `fit` function is calculated:

```
        if self.use_sample_weighting:
            self.model.fit(numpy.vstack((labeledX, unlabeledX)),
    labels, sample_weight=weights)
        else:
            self.model.fit(numpy.vstack((labeledX, unlabeledX)),
    labels)


        if self.verbose > 1:
            print "number of non-one soft labels: ", numpy.sum(self.
    bestsoftlbl != 1), ", balance:", numpy.sum(self.bestsoftlbl<0.5), " /
    ", len(self.bestsoftlbl)
            print "current likelihood: ", ll
```

Now that we've had a chance to understand the implementation of CPLE, let's get hands-on with an interesting dataset of our own! This time, we'll change things up by working with the University of Columbia's Million Song Dataset.

The central feature of this algorithm is feature analysis and metadata for one million songs. The data is preprepared and made up of natural and derived features. Available features include things such as the artist's name and ID, duration, loudness, time signature, and tempo of each song, as well as other measures including a crowd-rated danceability score and tags associated with the audio.

This dataset is generally labeled (via tags), but our objective in this case will be to generate genre labels for different songs based on the data provided. As the full million song dataset is a rather forbidding 300 GB, let's work with a 1% (1.8 GB) subset of 10,000 records. Furthermore, we don't particularly need this data as it currently exists; it's in an unhelpful format and a lot of the fields are going to be of little use to us.

The `10000_songs` dataset residing in the *Chapter 6*, *Text Feature Engineering* folder of our *Mastering Python Machine Learning* repository is a cleaned, prepared (and also rather large) subset of music data from multiple genres. In this analysis, we'll be attempting to predict genre from the genre tags provided as targets. We'll take a subset of tags as the labeled data used to kick-start our learning and will attempt to generate tags for unlabelled data.

In this iteration, we're going to raise our game as follows:

- Using more labeled data. This time, we'll use 1% of the total dataset size (100 songs), taken at random, as labeled data.
- Using an SVM with a linear kernel as our classifier, rather than the simple linear discriminant analysis we used with our naïve self-training implementation earlier in this chapter.

So, let's get started:

```
import sklearn.svm
import numpy as np
import random

from frameworks.CPLELearning import CPLELearningModel
from methods import scikitTSVM
from examples.plotutils import evaluate_and_plot

kernel = "linear"

songs = fetch_mldata("10000_songs")
X = songs.data
ytrue = np.copy(songs.target)
ytrue[ytrue==-1]=0

labeled_N = 20
ys = np.array([-1]*len(ytrue))
random_labeled_points = random.sample(np.where(ytrue == 0)[0],
labeled_N/2)+\
                        random.sample(np.where(ytrue == 1)[0],
labeled_N/2)
ys[random_labeled_points] = ytrue[random_labeled_points]
```

For comparison, we'll run a supervised SVM alongside our CPLE implementation. We'll also run the naïve self-supervised implementation, which we saw earlier in this chapter, for comparison:
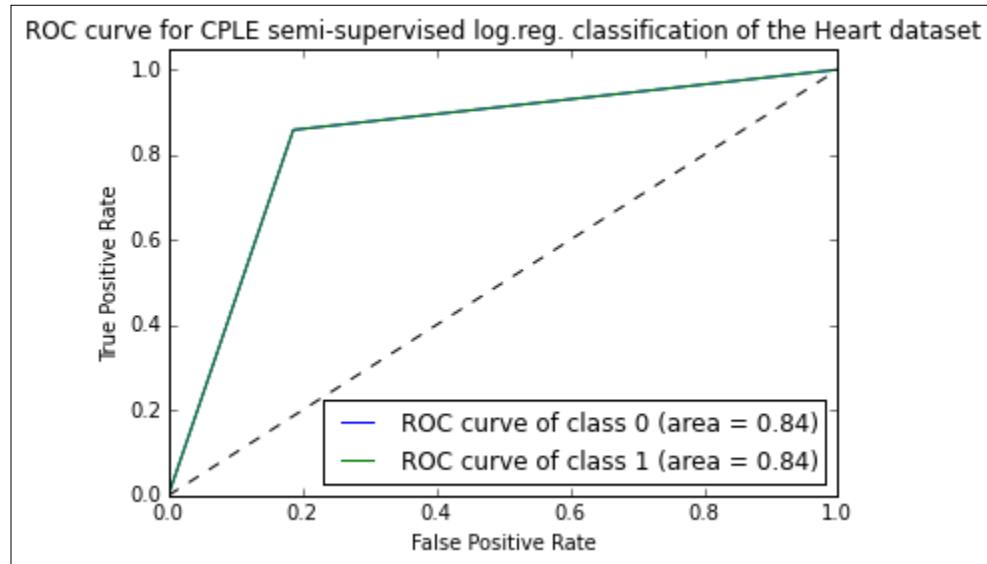
```
basemodel = SGDClassifier(loss='log', penalty='l1') # scikit logistic
regression
basemodel.fit(X[random_labeled_points, :], ys[random_labeled_points])
print "supervised log.reg. score", basemodel.score(X, ytrue)


ssmodel = SelfLearningModel(basemodel)
ssmodel.fit(X, ys)
print "self-learning log.reg. score", ssmodel.score(X, ytrue)


ssmodel = CPLELearningModel(basemodel)
ssmodel.fit(X, ys)
print "CPLE semi-supervised log.reg. score", ssmodel.score(X, ytrue)
```

The results that we obtain on this iteration are very strong:

```
# supervised log.reg. score 0.698
# self-learning log.reg. score 0.825
# CPLE semi-supervised log.reg. score 0.833
```

The CPLE semi-supervised model succeeds in classifying with 84% accuracy, a score comparable to human estimation and over 10% higher than the naïve semi-supervised implementation. Notably, it also outperforms the supervised SVM.

Further reading

A solid place to start understanding Semi-supervised learning methods is Xiaojin Zhu's very thorough literature survey, available at `http://pages.cs.wisc.edu/~jerryzhu/pub/ssl_survey.pdf`.

I also recommend a tutorial by the same author, available in the slide format at `http://pages.cs.wisc.edu/~jerryzhu/pub/sslicml07.pdf`.

The key paper on Contastive Pessimistic Likelihood Estimation is Loog's 2015 paper `http://arxiv.org/abs/1503.00269`.

This chapter made a reference to the distinction between generative and discriminative models. A couple of relatively clear explanations of the distinction between generative and discriminative algorithms are provided by Andrew Ng (`http://cs229.stanford.edu/notes/cs229-notes2.pdf`) and Michael Jordan (`http://www.ics.uci.edu/~smyth/courses/cs274/readings/jordan_logistic.pdf`).

For readers interested in Bayesian statistics, Allen Downey's book, *Think Bayes*, is a marvelous introduction (and one of my all-time favorite statistics books): `https://www.google.co.uk/#q=think+bayes`.

For readers interested in learning more about gradient descent, I recommend Sebastian Ruder's blog at `http://sebastianruder.com/optimizing-gradient-descent/`.

For readers interested in going a little deeper into the internals of conjugate descent, Jonathan Shewchuk's introduction provides clear and enjoyable definitions for a number of key concepts at `https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`.

Summary

In this chapter, we tapped into a very powerful but lesser known paradigm in machine learning—semi-supervised learning. We began by exploring the underlying concepts of transductive learning and self-training, and improved our understanding of the latter class of techniques by working with a naïve self-training implementation.

We quickly began to see weaknesses in self-training and looked for an effective solution, which we found in the form of CPLE. CPLE is a very elegant and highly applicable framework for semi-supervised learning that makes no assumptions beyond those of the classifier that it uses as a base model. In return, we found CPLE to consistently offer performance in excess of naïve semi-supervised and supervised implementations, at minimal risk. We've gained a significant amount of understanding regarding one of the most useful recent developments in machine learning.

In the next chapter, we'll begin discussing data preparation skills that significantly increase the effectiveness of all of the models that we've previously discussed.

6

Text Feature Engineering

Introduction

In preceding chapters, we've spent time assessing powerful techniques that enable the analysis of complex or challenging data. However, for the most difficult problems, the right technique will only get you so far.

The persistent challenge that deep learning and supervised learning try to solve for is that finding solutions often requires multiple big investments from the team in question. Under the old paradigm, one often has to perform specific preparation tasks, requiring time, specialist skills, and knowledge. Often, even the techniques used were domain-specific and/or data type-specific. This process, via which features are derived, is referred to as feature engineering.

Most of the deep learning algorithms which we've studied so far are intended to help find ways around needing to perform extensive feature engineering. However, at the same time, feature engineering continues to be seen as a hugely important skill for top-level ML practitioners. The following quotes come from leading Kaggle competitors, via David Kofoed Wind's contribution to the Kaggle blog:

> *"The features you use influence more than everything else the result. No algorithm alone, to my knowledge, can supplement the information gain given by correct feature engineering."*

> *– (Luca Massaron)*

> *"Feature engineering is certainly one of the most important aspects in Kaggle competitions and it is the part where one should spend the most time on. There are often some hidden features in the data which can improve your performance by a lot and if you want to get a good place on the leaderboard you have to find them. If you screw up here you mostly can't win anymore; there is always one guy who finds all the secrets. However, there are also other important parts, like how you formulate the problem. Will you use a regression model or classification model or even combine both or is some kind of ranking needed. This, and feature engineering, are crucial to achieve a good result in those competitions. There are also some competitions where (manual) feature engineering is not needed anymore; like in image processing competitions. Current state of the art deep learning algorithms can do that for you."*

> *– (Josef Feigl)*

There are a few key themes here; feature engineering is powerful and even a very small amount of feature engineering can have a big impact on one's classifiers. It is also frequently necessary to employ feature engineering techniques if one wishes to deliver the best possible results. Maximising the effectiveness of your machine learning algorithms requires a certain amount of both domain-specific and data type-specific knowledge (secrets).

One more quote:

> *"For most Kaggle competitions the most important part is feature engineering, which is pretty easy to learn how to do."*

> *– (Tim Salimans)*

Tim's not wrong; most of what you'll learn in this chapter is intuitive, effective tricks, and transformations. This chapter will introduce you to a few of the most effective and commonly-used preparation techniques applied to text and time series data, drawing from NLP and financial time series applications. We'll walk through how the techniques work, what one should expect to see, and how one can diagnose whether they're working as desired.

Text feature engineering

In preceding sections, we've discussed some of the methods by which we might take a dataset and extract a subset of valuable features. These methods have broad applicability but are less helpful when dealing with non-numerical/non-categorical data, or data that cannot be easily translated into numerical or categorical data. In particular, we need to apply different techniques when working with text data.

The techniques that we'll study in this section fall into two main categories—cleaning techniques and feature preparation techniques. These are typically implemented in roughly that order and we'll study them accordingly.

Cleaning text data

When we work with natural text data, a different set of approaches apply. This is because in real-world contexts, the idea of a naturally clean text dataset is pretty unsafe; text data is rife with misspellings, non-dictionary constructs like emoticons, and in some cases, HTML tagging. As such, we need to be very thorough with our cleaning.

In this section, we'll work with a number of effective text-cleaning techniques, using a pretty gnarly real-world dataset. Specifically, we'll be using the Impermium dataset from a 2012 Kaggle competition, where the competition's goal was to create a model which accurately detects insults in social commentary.

Yes, I do mean Internet troll detection.

Let's get started!

Text cleaning with BeautifulSoup

Our first step should be manually checking the input data. This is pretty critical; with text data, one needs to try and understand what issues exist in the data initially so as to identify the cleaning needed.

It's kind of painful to read through a dataset full of hateful Internet commentary, so here's an example entry:

ID	Date	Comment
132	20120531031917Z	`"""\xa0@Flip\xa0how are you not ded"""`

We have an ID field and date field which don't seem to need much work. The text fields, however, are quite challenging. From this one case, we can already see misspelling and HTML inclusion. Furthermore, many entries in the dataset contain attempts to bypass swear filtering, usually by including a space or punctuation element mid-word. Other data quality issues include multiple vowels (to extend a word), non-ascii characters, hyperlinks... the list goes on.

One option for cleaning this dataset is to use regular expressions, which run over the input data to scrub out data quality issues. However, the quantity and variety of problem formats make it impractical to use a regex-based approach, at least to begin with. We're likely both to miss a lot of cases and also to misjudge the amount of preparation needed, leading us to clean too aggressively, or not aggressively enough; in specific terms we risk cutting into real text content or leaving parts of tags in place. What we need is a solution that will wash out the majority of common data quality problems to begin with so that we can focus on the remaining issues with a script-based approach.

Enter `BeautifulSoup`. `BeautifulSoup` is a very powerful text cleaning library which can, among other things, remove HTML markup. Let's take a look at this library in action on our troll data:

```python
from bs4 import BeautifulSoup
import csv

trolls = []
with open('trolls.csv',  'rt') as f:
    reader = csv.DictReader(f)
    for line in reader:
        trolls.append(BeautifulSoup(str(line["Comment"]), "html.
parser"))

print(trolls[0])

eg = BeautifulSoup(str(trolls), "html.parser")

print(eg.get_text())
```

ID	Date	Comments
132	20120531031917Z	@Flip how are you not ded

As we can see, we've already made headway on improving the quality of our text data. Yet, it's also clear from these examples that there's a lot of work left to do! As discussed, let's move on to using regular expressions to help further clean and tokenize our data.

Managing punctuation and tokenizing

Tokenisation is the process of creating a set of tokens from a stream of text. Many tokens are words, while others might be character sets (such as smilies or other punctuation strings, for example, `????????`).

Now that we've removed a lot of the HTML ugliness from our initial dataset, we can take steps to further improve the cleanliness of our text data. To do this, we'll leverage the `re` module, which allows us to use operations over regular expressions, such as substring replacement. We'll perform a series of operations over our input text on this pass, which mostly focus on replacing variable or problematic text elements with tokens. Let's begin with a simple example, replacing e-mail addresses with an `_EM` token:

```
text = re.sub(r'[\w\-][\w\-\.]+@[\w\-][\w\-\.]+[a-zA-Z]{1,4}', '_EM',
text)
```

Similarly, we can remove URLs, replacing them with the `_U` token:

```
text = re.sub(r'\w+:\/\/\S+', r'_U', text)
```

We can automatically remove extra or problematic whitespace and newline characters, hyphens, and underscores. In addition, we'll begin managing the problem of multiple characters, often used for emphasis in informal conversation. Extended series of punctuation characters are encoded here using codes such as `_BQ` and `BX`; these longer tags are used as a means of differentiating from the more straightforward `_Q` and `_X` tags (which refer to the use of a question mark and exclamation mark, respectively).

We can also use regular expressions to manage extra letters; by cutting down such strings to two characters at most, we're able to reduce the number of combinations to a manageable amount and tokenize that reduced group using the `_EL` token:

```
# Format whitespaces
text = text.replace('"', ' ')
text = text.replace('\'', ' ')
text = text.replace('_', ' ')
text = text.replace('-', ' ')
text = text.replace('\n', ' ')
text = text.replace('\\n', ' ')
text = text.replace('\'', ' ')
text = re.sub(' +',' ', text)
text = text.replace('\'', ' ')


#manage punctuation
text = re.sub(r'([^!\?])(\?{2,})(\Z|[^!\?])', r'\1 _BQ\n\3', text)
text = re.sub(r'([^\.])(\.{2,})', r'\1 _SS\n', text)
text = re.sub(r'([^!\?])(\?|!){2,}(\Z|[^!\?])', r'\1 _BX\n\3', text)
text = re.sub(r'([^!\?])\?(\Z|[^!\?])', r'\1 _Q\n\2', text)
text = re.sub(r'([^!\?])!(\Z|[^!\?])', r'\1 _X\n\2', text)
```

```
text = re.sub(r'([a-zA-Z])\1\1+(\w*)', r'\1\1\2 _EL', text)
text = re.sub(r'([a-zA-Z])\1\1+(\w*)', r'\1\1\2 _EL', text)
text = re.sub(r'(\w+)\.(\w+)', r'\1\2', text)
text = re.sub(r'[^a-zA-Z]','', text)
```

Next, we want to begin creating other tokens of interest. One of the more helpful indicators available is the _SW token for swearing. We'll also use regular expressions to help identify and tokenize smileys into one of four buckets; big and happy smileys (_BS), small and happy ones (_S), big and sad ones (_BF), and small and sad ones (_F):

```
text = re.sub(r'([#%&\*\$]{2,})(\w*)', r'\1\2 _SW', text)
```

```
text = re.sub(r' [8x;:=]-?(?:\)|\}|\]|>){2,}', r' _BS', text)
text = re.sub(r' (?:[;:=]-?[\)\}\]d>])|(?:<3)', r' _S', text)
text = re.sub(r' [x:=]-?(?:\(|\[|\||\\|/|\{|<){2,}', r' _BF', text)
text = re.sub(r' [x:=]-?[\(\[\|\\/\{<]', r' _F', text)
```

> Smileys are complicated by the fact that their uses change frequently; while this series of characters is reasonably current, it's by no means complete; for example, see emojis for a range of non-ascii representations. For several reasons, we'll be removing non-ascii text from this example (a similar approach is to use a dictionary to force compliance), but both approaches have the obvious drawback that they remove cases from the dataset, meaning that any solution will be imperfect. In some cases, this approach may lead to the removal of a substantial amount of data. In general, then, it's prudent to be aware of the general challenge around character-based images in text content.

Following this, we want to begin splitting text into phrases. This is a simple application of str.split, which enables the input to be treated as a vector of words (words) rather than as long strings (re):

```
phrases = re.split(r'[;:\.()\n]', text)
phrases = [re.findall(r'[\w%\*&#]+', ph) for ph in phrases]
phrases = [ph for ph in phrases if ph]

words = []

for ph in phrases:
        words.extend(ph)
```

This gives us the following:

ID	Date	Comments
132	20120531031917Z	[['Flip', 'how', 'are', 'you', 'not', 'ded']]

Next, we perform a search for single-letter sequences. Sometimes, for emphasis, Internet communication involves the use of spaced single-letter chains. This may be attempted as a method of avoiding curse word detection:

```
tmp = words
words = []
new_word = ''
for word in tmp:
    if len(word) == 1:
        new_word = new_word + word
    else:
        if new_word:
            words.append(new_word)
            new_word = ''
        words.append(word)
```

So far, then, we've gone a long way toward cleaning and improving the quality of our input data. There are still outstanding issues, however. Let's reconsider the example we began with, which now looks like the following:

ID	Date	Words
132	20120531031917Z	['_F', 'how', 'are', 'you', 'not', 'ded']

Much of our early cleaning has passed this example by, but we can see the effect of vectorising the sentence content as well as the now-cleaned HTML tags. We can also see that the emote used has been captured via the _F tag. When we look at a more complex test case, we see even more substantial change results:

Raw	Cleaned and split
GALLUP DAILY\nMay 24-26, 2012 \ u2013 Updates daily at 1 p.m. ET; reflects one-day change\ nNo updates Monday, May 28; next update will be Tuesday, May 29.\nObama Approval48%-\nObama Disapproval45%-1\nPRESIDENTIAL ELECTION\nObama47%-\nRomney45%-\ n7-day rolling average\n\n It seems the bump Romney got is over and the president is on his game.	['GALLUP', 'DAILY', 'May', 'u', 'Updates', 'daily', 'pm', 'ET', 'reflects', 'one', 'day', 'change', 'No', 'updates', 'Monday', 'May', 'next', 'update', 'Tuesday', 'May', 'Obama', 'Approval', 'Obama', 'Disapproval', 'PRESIDENTIAL', 'ELECTION', 'Obama', 'Romney', 'day', 'rolling', 'average', 'It', 'seems', 'bump', 'Romney', 'got', 'president', 'game']

However, there are two significant problems still obvious in both examples. In the first case, we have a misspelled word; we need to find a way to eliminate this. Secondly, a lot of the words in both examples (for example. are, pm) aren't terribly informative in and of themselves. The problem we find, particularly for shorter text samples, is that what's left after cleaning may contain only one or two meaningful terms. If these terms are not terribly common in the corpus as a whole, it can prove to be very difficult to train a classifier to recognise these terms' significance.

Tagging and categorising words

I expect that we all know that English language words come in several types—nouns, verbs, adverbs, and so on. These are commonly referred to as **parts of speech**. If we know that a certain word is an adjective, as opposed to a verb or stop word (such as a, the, or of), we can treat it differently or more importantly, our algorithm can!

If we can perform part of speech tagging by identifying and encoding word classes as categorical variables, we're able to improve the quality of our data by retaining only the valuable content. The full range of text tagging options and techniques is too broad to be effectively covered in one section of this chapter, so we'll look at a subset of the applicable tagging techniques. Specifically, we'll focus on n-gram tagging and backoff taggers, a pair of complimentary techniques that allow us to create powerful recursive tagging algorithms.

We'll be using a Python library called the **Natural Language Toolkit** (**NLTK**). NLTK offers a wide array of functionality and we'll be relying on it at several points in this chapter. For now, we'll be using NLTK to perform tagging and removal of certain word types. Specifically, we'll be filtering out stop words.

To answer the obvious first question (why eliminate stop words?), it tends to be true that stop words add little to nothing to most text analysis and can be responsible for a degree of noise and training variance. Fortunately, filtering stop words is pretty straightforward. We'll simply import NLTK, download and import the dictionaries, then perform a scan over all words in our pre-existing word vector, removing any stop words found:

```
import nltk
nltk.download()
from nltk.corpus import stopwords

words = [w for w in words if not w in stopwords.words("english")]
```

I'm sure you'll agree that this was pretty straightforward! Let's move on to discuss more NLTK functionality, specifically, tagging.

Tagging with NLTK

Tagging is the process of identifying parts of speech, as we described previously, and applying tags to each term.

In its simplest form, tagging can be as straightforward as applying a dictionary over our input data, just as we did previously with stopwords:

```
tagged = ntlk.word_tokenize(words)
```

However, even brief consideration will make it obvious that our use of language is a lot more complicated than this allows. We may use a word (such as ferry) as one of several parts of speech and it may not be straightforward to decide how to treat each word in every utterance. A lot of the time, the correct tag can only be understood contextually given the other words and their positioning within the phrase.

Thankfully, we have a number of useful techniques available to help us solve linguistic challenges.

Sequential tagging

A sequential tagging algorithm is one that works by running through the input dataset, left-to-right and token-by-token (hence sequential!), tagging each token in succession. The decision over which token to assign is made based on that token, the tokens that preceded it, and the predicted tags for those preceding tokens.

In this section, we'll be using an **n-gram tagger**. An n-gram tagger is a type of sequential tagger, which is pretrained to identify appropriate tags. The n-gram tagger takes *(n-1)-many* preceding POS tags and the current token into consideration in producing a tag.

> For clarity, an n-gram is the term used for a contiguous sequence of n-many elements from a given set of elements. This may be a contiguous sequence of letters, words, numerical codes (for example, for state changes), or other elements. N-grams are widely used as a means of capturing the conjunct meaning of sets of elements—be those phrases or encoded state transitions—using n-many elements.

The simplest form of n-gram tagger is one where *n = 1*, referred to as a **unigram tagger**. A unigram tagger operates quite simply, by maintaining a conditional frequency distribution for each token. This conditional frequency distribution is built up from a training corpus of terms; we can implement training using a helpful train method belonging to the `NgramTagger` class in NLTK. The tagger assumes that the tag which occurs most frequently for a given token in a given sequence is likely to be the correct tag for that token. If the term **carp** is in the training corpus as a noun four times and as a verb twice, a unigram tagger will assign the noun tag to any token whose type is carp.

This might suffice for a first-pass tagging attempt but clearly, a solution that only ever serves up one tag for each set of homonyms isn't always going to be ideal. The solution we can tap into is using n-grams with a larger value of *n*. With *n = 3* (a **trigram tagger**), for instance, we can see how the tagger might more easily distinguish the input *He tends to carp on a lot* from *He caught a magnificent carp*!

However, once again there is a trade-off here between accuracy of tagging and ability to tag. As we increase *n*, we're creating increasingly long n-grams which become increasingly rare. In a very short time, we end up in a situation where our n-grams are not occurring in the training data, causing our tagger to be unable to find any appropriate tag for the current token!

In practice, we find that what we need is a set of taggers. We want our most reliably accurate tagger to have the first shot at trying to tag a given dataset and, for any case that fails, we're comfortable with having a more reliable but potentially less accurate tagger have a try.
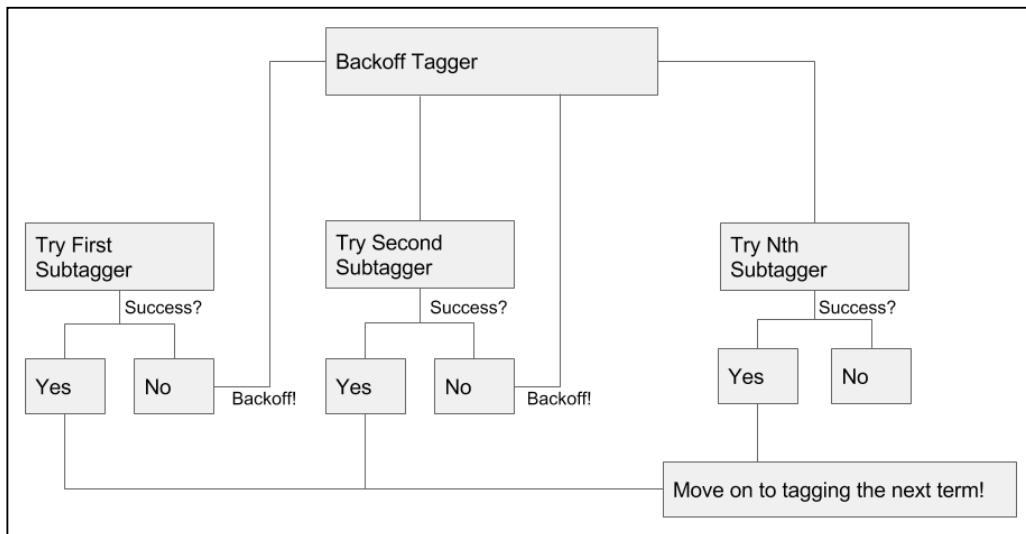
Happily, what we want already exists in the form of backoff tagging. Let's find out more!

Backoff tagging

Sometimes, a given tagger may not perform reliably. This is particularly common when the tagger has high accuracy demands and limited training data. At such times, we usually want to build an ensemble structure that lets us use several taggers simultaneously.
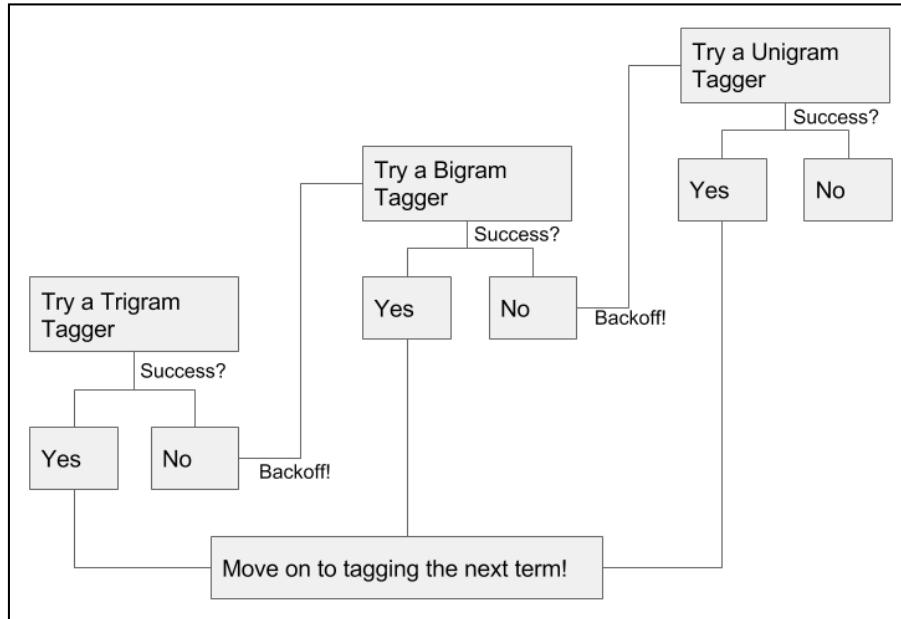
To do this, we create a distinction between two types of taggers: **subtaggers** and **backoff taggers**. Subtaggers are taggers like the ones we saw previously, **sequential** and **Brill taggers**. Tagging structures may contain one or multiple of each kind of tagger.

If a subtagger is unable to determine a tag for a given token, then a backoff tagger may be referred to instead. A backoff tagger is specifically used to combine the results of an ensemble of (one or more) subtaggers, as shown in the following example diagram:

In simple implementations, the backoff tagger will simply poll the subtaggers in order, accepting the first none-null tag provided. If all subtaggers return null for a given token, the backoff tagger will assign a none tag to that token. The order can be determined.

Backoffs are typically used with multiple subtaggers of different types; this enables a data scientist to harness the benefits of multiple types of tagger simultaneously. Backoffs may refer to other backoffs as needed, potentially creating highly redundant or sophisticated tagging structures:

In general terms, backoff taggers provide redundancy and enable you to use multiple taggers in a composite solution. To solve our immediate problem, let's implement a nested series of n-gram taggers. We'll start with a trigram tagger, which will use a bigram tagger as its backoff tagger. If neither of these taggers has a solution, we'll have a unigram tagger as an additional backoff. This can be done very simply, as follows:

```
brown_a = nltk.corpus.brown.tagged_sents(categories= 'a')

tagger = None
for n in range(1,4):
  tagger = NgramTagger(n, brown_a, backoff = tagger)

words  = tagger.tag(words)
```

Creating features from text data

Once we've engaged in well-thought-out text cleaning practices, we need to take additional steps to ensure that our text becomes useful features. In order to do this, we'll look at another set of staple techniques in NLP:

- Stemming
- Lemmatising
- Bagging using random forests

Stemming

Another challenge when working with linguistic datasets is that multiple word forms exist for many word stems. For example, the root dance is the stem of multiple other words—dancing, dancer, dances, and so on. By finding a way to reduce this plurality of forms into stems, we find ourselves able to improve our n-gram tagging and apply new techniques such as lemmatisation.

The techniques that enable us to reduce words to their stems are called stemmers. Stemmers work by parsing words as consonant/vowel strings and applying a series of rules. The most popular stemmer is the **porter stemmer**, which works by performing the following steps;

1. Simplifying the range of suffixes by reducing (for example, *ies* becomes *i*) to a smaller set.
2. Removing suffixes in several passes, with each pass removing a set of suffix types (for example, past particple or plural suffixes such as ousness or alism).
3. Once all suffixes are removed, cleaning up word endings by adding 'e's where needed (for example, ceas becomes cease).
4. Removing double 'l's.

The porter stemmer works very effectively. In order to understand exactly how well it works, let's see it in action!

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

stemmer.stem(words)
```

The output of this `stemmer`, as demonstrated on our pre-existing example, is the root form of the word. This may be a real word, or it may not; dancing, for instance, becomes danci. This is okay, but it's not really ideal. We can do better than this!

To consistently reach a real word form, let's apply a slightly different technique, lemmatisation. Lemmatisation is a more complex process to determine word stems; unlike porter stemming, it uses a different normalisation process for different parts of speech. Unlike Porter Stemming it also seeks to find actual roots for words. Where a stem does not have to be a real word, a lemma does. Lemmatization also takes on the challenge of reducing synonyms down to their roots. For example, where a stemmer might turn the term books into the term book, it isn't equipped to handle the term tome. A lemmatizer can process both books and tome, reducing both terms to book.

As a necessary prerequisite, we need the POS for each input token. Thankfully, we've already applied a POS tagger and can work straight from the results of that process!

```
from nltk.stem import PorterStemmer, WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

words = lemmatizer.lemmatize(words, pos = 'pos')
```

The output is now what we'd expect to see:

Source Text	Post-lemmatisation
The laughs you two heard were triggered by memories of his own high-flying exits off moving beasts	['The', 'laugh', 'two', 'hear', 'trigger', 'memory', 'high', 'fly', 'exit', 'move', 'beast']

We've now successfully stemmed our input text data, massively improving the effectiveness of lookup algorithms (such as many dictionary-based approaches) in handling this data. We've removed stop words and tokenized a range of other noise elements with regex methods. We've also removed any HTML tagging. Our text data has reached a reasonably processed state. There's one more linchpin technique that we need to learn, which will let us generate features from our text data. Specifically, we can use bagging to help quantify the use of terms.

Let's find out more!

Bagging and random forests

Bagging is part of a family of techniques that may collectively be referred to as subspace methods. There are several forms of method, with each having a separate name. If we draw random subsets from the sample cases, then we're performing pasting. If we're sampling from cases with replacement, it's referred to as bagging. If instead of drawing from cases, we work with a subset of features, then we're performing attribute bagging. Finally, if we choose to draw from both sample cases and features, we're employing what's known as a **random patches** technique.

The feature-based techniques, attribute bagging, and Random Patch methods are very valuable in certain contexts, particularly very high-dimensional ones. Medical and genetics contexts both tend to see a lot of extremely high-dimensional data, so feature-based methods are highly effective within those contexts.

In NLP contexts, it's common to work with bagging specifically. In the context of linguistic data, what we'll be dealing with is properly called a bag of words. A bag of words is an approach to text data preparation that works by identifying all of the distinct words (or tokens) in a dataset and then counting their occurrence in each sample. Let's begin with a demonstration, performed over a couple of example cases from our dataset:

ID	Date	Words
132	20120531031917Z	['_F', 'how', 'are', 'you', 'not', 'ded']
69	20120531173030Z	['you', 'are', 'living', 'proof', 'that', 'bath', 'salts', 'effect', 'thinking']

This gives us the following 12-part list of terms:

```
[
  "_F"
  "how"
  "are"
  "you"
  "not"
  "ded"
  "living"
  "proof"
  "that"
  "bath"
```

```
    "salts"

    "effect"

    "thinking"

]
```

Using the indices of this list, we can create a 12-part vector for each of the preceding sentences. This vector's values are filled by traversing the preceding list and counting the number of times each term occurs for each sentence in the dataset. Given our pre-existing example sentences and the list we created from them, we end up creating the following bags:

ID	Date	Comment	Bag of words
132	20120531031917Z	_F how are you not ded	[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
69	20120531173030Z	you are living proof that bath salts effect thinking	[0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1]

This is the core of a bag of words implementation. Naturally, once we've translated the linguistic content of text into numerical vectors, we're able to start using techniques that add sophistication to how we use this text in classification.

One option is to use weighted terms. We can use a term weighting scheme to modify the values within each vector so that terms that are indicative or helpful for classification are emphasized. Weighting schemes may be straightforward masks, such as a binary mask that indicates presence versus absence.

Binary masking can be useful if certain terms are used much more frequently than normal; in such cases, specific scaling (for example, log-scaling) may be needed if a binary mask is not used. At the same time, though, frequency of term use can be informative (it may indicate emphasis, for instance) and the decision over whether to apply a binary mask is not always made simply.

Another weighting option is term frequency-inverse document frequency, or tf-idf. This scheme compares frequency of usage within a specific sentence and the dataset as a whole and uses values that increase if a term is used more frequently within a given sample than within the whole corpus.

Variations on tf-idf are frequently used in text mining contexts, including search engines. Scikit-learn provides a tf-idf implementation, `TfidfVectoriser`, which we'll shortly use to employ tf-idf for ourselves.

Now that we have an understanding of the theory behind bag of words and can see the range of technical options available to us once we develop vectors of word use, we should discuss how a bag of words implementation can be undertaken. Bag of words can be easily applied as a wrapper to a familiar model. While in general, subspace methods may use any of a range of base models (SVMs and linear regression models are common), it is very common to use random forests in a bag of words implementation, wrapping up preparation and learning into a concise script. In this case, we'll employ bag of words independently for now, saving classification via a random forest implementation for the next section!

> While we'll discuss random forests in greater detail in *Chapter 8, Ensemble Methods*, (which describes the various types of ensemble that we can create), it is helpful for now to note that a random forest is a set of decision trees. They are powerful ensemble models that are created either to run in parallel (yielding a vote or other net outcome) or boost one another (by iteratively adding a new tree to model the parts of the solution that the pre-existing set of trees couldn't model well).
>
> Due to the power and ease of use of random forests, they are commonly used as benchmarking algorithms.

The process of implementing bag of words is, again, fairly straightforward. We initialize our bagging tool (matter-of-factly referred to as a vectorizer). Note that for this example, we're putting a limit on the size of the feature vector. This is largely to save ourselves some time; each document must be compared against each item in the feature list, so when we get to running our classifier this could take a little while!

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(analyzer = "word",   \
                             tokenizer = None,     \
                             preprocessor = None, \
                             stop_words = None,    \
                             max_features = 5000)
```

Our next step is to fit the vectorizer on our word data via `fit_transform`; as part of the fitting process, our data is transformed into feature vectors:

```
train_data_features = vectorizer.fit_transform(words)

train_data_features = train_data_features.toarray()
```

This completes the pre-processing of our text data. We've taken this dataset through a full battery of text mining techniques, walking through the theory and reasoning behind each technique as well as employing some powerful Python scripts to process our test dataset.We're in a good position now to take a crack at Kaggle's insult detection challenge!

Testing our prepared data

So, now that we've done some initial preparation of the dataset, let's give the real problem a shot and see how we do. To help set the scene, let's consider Impermium's guidance and data description:

This is a single-class classification problem. The label is either 0 meaning a neutral comment, or 1 meaning an insulting comment (neutral can be considered as not belonging to the insult class. Your predictions must be a real number in the range [0,1] where 1 indicates 100% confident prediction that comment is an insult.

- *We are looking for comments that are intended to be insulting to a person who is a part of the larger blog/forum conversation.*

- *We are NOT looking for insults directed to non-participants (such as celebrities, public figures etc.).*

- *Insults could contain profanity, racial slurs, or other offensive language. But often times, they do not.*

- *Comments which contain profanity or racial slurs, but are not necessarily insulting to another person are considered not insulting.*

- *The insulting nature of the comment should be obvious, and not subtle.*

- *There may be a small amount of noise in the labels as they have not been meticulously cleaned. However, contestants can be confident the error in the training and testing data is < 1%.*

Contestants should also be warned that this problem tends to strongly overfit. The provided data is generally representative of the full test set, but not exhaustive by any measure. Impermium will be conducting final evaluations based on an unpublished set of data drawn from a wide sample.

This is pretty nice guidance, in that it raises two particular points of caution. The desired score is the **area under the curve** (**AUC**), which is a measure that is very sensitive both to false positives and to incorrect negative results (specificity and sensitivity).

The guidance clearly states that continuous predictions are desired rather than binary *0/1* outputs. This becomes critically important when using AUC; even a small amount of incorrect predictions given will radically decrease one's score if you only use categorical values. This suggests that rather than using the `RandomForestClassifier` algorithm, we'll want to use the `RandomForestRegressor`, a regression-focused alternative, and then rescale the results between zero and one.

Real Kaggle contests are run in a much more challenging and realistic environment—one where the correct solution is not available. In *Chapter 8*, *Ensemble Methods*, we'll explore how top data scientists react and thrive in such environments. For now, we'll take advantage of having the ability to confirm whether we're doing well on the test dataset. Note that this advantage also presents a risk; if the problem overfits strongly, we'll need to be disciplined to ensure that we're not overtraining on the test data!

In addition, we also have the benefit of being able to see how well real contestants did. While we'll save the real discussion for *Chapter 8*, *Ensemble Methods*, it's reasonable to expect each highly-ranking contestant to have submitted quite a large number of failed attempts; having a benchmark will help us tell whether we're heading in the right direction.

Specifically, the top 14 participants on the private (test) leaderboard managed to reach an AUC score of over *0.8*. The top scorer managed a pretty impressive *0.84*, while over half of the 50 teams who entered scored above *0.77*.

As we discussed earlier, let's begin with a random forest regression model.

> A random forest is an ensemble of decision trees.
>
> While a single decision tree is likely to suffer from variance- or bias-related issues, random forests are able to use a weighted average over multiple parallel trials to balance the results of modeling.
>
> Random forests are very straightforward to apply and are a good first-pass technique for a new data challenge; applying a random forest classifier to the data early on enables you to develop a good understanding as to what initial, baseline classification accuracy will look like as well as giving valuable insight into how the classification boundaries were formed; during the initial stages of working with a dataset, this kind of insight is invaluable.
>
> Scikit-learn provides `RandomForestClassifier` to enable the easy application of a random forest algorithm.

For this first pass, we'll use 100 trees; increasing the number of trees can improve classification accuracy but will take additional time. Generally speaking, it's sensible to attempt fast iteration in the early stages of model creation; the faster you can repeatedly run your model, the faster you can learn what your results are looking like and how to improve them!

We begin by initializing and training our model:

```
trollspotter = RandomForestRegressor(n_estimators = 100, max_depth =
10, max_features = 1000)

y = trolls["y"]

trollspotted = trollspotter.fit(train_data_features, y)
```

We then grab the test data and apply our model to predict a score for each test case. We rescale these scores using a simple stretching technique:

```
moretrolls = pd.read_csv('moretrolls.csv', header=True, names=['y',
'date', 'Comment', 'Usage'])
moretrolls["Words"] = moretrolls["Comment"].apply(cleaner)

y = moretrolls["y"]

test_data_features = vectorizer.fit_transform(moretrolls["Words"])
test_data_features = test_data_features.toarray()

pred = pred.predict(test_data_features)
pred = (pred - pred.min())/(pred.max() - pred.min())
```

Finally, we apply the `roc_auc` function to calculate an AUC score for the model:

```
fpr, tpr, _ = roc_curve(y, pred)
roc_auc = auc(fpr, tpr)
print("Random Forest benchmark AUC score, 100 estimators")
print(roc_auc)
```

As we can see, the results are definitely not at the level that we want them to be at:

Random Forest benchmark AUC score, 100 estimators

0.537894912105

Thankfully, we have a number of options that we can try to configure here:

- Our approach to how we work with the input (preprocessing steps and normalisation)
- The number of estimators in our random forest
- The classifier we choose to employ
- The properties of our bag of words implementation (particularly the maximum number of terms)
- The structure of our n-gram tagger

On our next pass, let's adjust the size of our bag of words implementation, increasing the term cap from a slightly arbitrary 5,000 to anywhere up to 8,000 terms; rather than picking just one value, we'll run over a range and see what we can learn. We'll also increase the number of trees to a more reasonable number (in this case, we stepped up to `1000`):

Random Forest benchmark AUC score, 1000 estimators

0.546439310772

These results are slightly better than the previous set, but not dramatically so. They're definitely a fair distance from where we want to be! Let's go further and set up a different classifier. Let's try a fairly familiar option—the SVM. We'll set up our own SVM object to work with:

```
class SVM(object):

    def __init__(self, texts, classes, nlpdict=None):

        self.svm = svm.LinearSVC(C=1000, class_weight='auto')
        if nlpdict:
            self.dictionary = nlpdict
        else:
            self.dictionary = NLPDict(texts=texts)
        self._train(texts, classes)

    def _train(self, texts, classes):
        vectors = self.dictionary.feature_vectors(texts)
        self.svm.fit(vectors, classes)

    def classify(self, texts):
        vectors = self.dictionary.feature_vectors(texts)
```

```
            predictions = self.svm.decision_function(vectors)
            predictions = p.transpose(predictions)[0:len(predictions)]
            predictions = predictions / 2 + 0.5
            predictions[predictions > 1] = 1
            predictions[predictions < 0] = 0
            return predictions
```

While the workings of SVM are almost impenetrable to human assessment, as an algorithm it operates effectively, iteratively translating the dataset into multiple additional dimensions in order to create complex hyperplanes at optimal class boundaries. It isn't a huge surprise, then, to see that the quality of our classification has increased:

SVM AUC score

0.625245653817
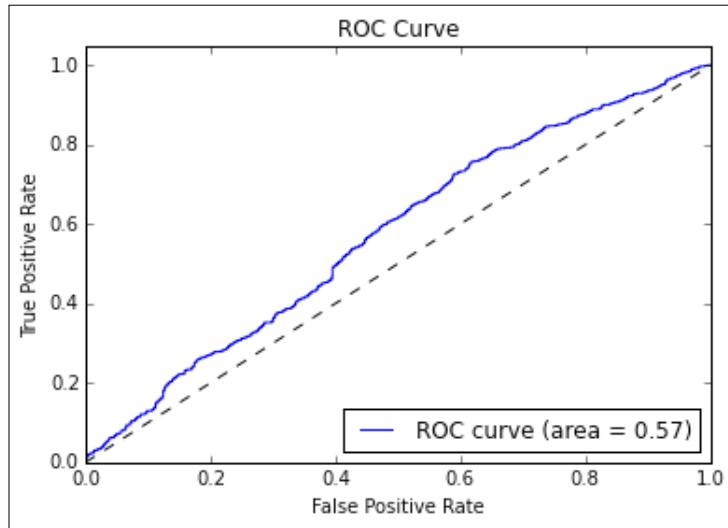
Perhaps we're not getting enough visibility into what's happening with our results. Let's try shaking things up with a different take on performance measurement. Specifically, let's look at the difference between the model's label predictions and actual targets to see whether the model is failing more frequently with certain types of input.

So we've taken our prediction quite far. While we still have a number of options on the table, it's worth considering the use of a more sophisticated ensemble of models as being a solid option. In this case, leveraging multiple models instead of just one can enable us to obtain the relative advantages of each. To try out an ensemble against this example, run the `score_trolls_blendedensemble.py` script.

> This ensemble is a blended/stacked ensemble. We'll be spending more time discussing how this ensemble works in *Chapter 8*, *Ensemble Methods*!

Plotting our results, we can see that performance has improved, but by significantly less than we'd hoped:

We're clearly having some issues with building a model against this data, but at this point, there isn't a lot of value in throwing a more developed model at the problem. We need to go back to our features and aim to extend the feature set.

At this point, it's worth taking some pointers from one of the most successful entrants of this particular Kaggle contest. In general, top-scoring entries tend to be developed by finding all of the tricks around the input data. The second-place contestant in the official Kaggle contest that this dataset was drawn from was a user named tuzzeg. This contestant provided a usable code repository at `https://github.com/tuzzeg/detect_insults`.

Tuzzeg's implementation differs from ours by virtue of much greater thoroughness. In addition to the basic features that we built using POS tagging, he employed POS-based bigrams and trigrams as well as subsequences (created from sliding windows of N-many terms). He worked with n-grams up to 7-grams and created character n-grams of lengths 2, 3, and 4.

Furthermore, tuzzeg took the time to create two types of composite model, both of which were incorporated into his solution—sentence level and ranking models. Ranking took our rationalization around the nature of the problem a step further by turning the cases in our data into ranked continuous values.

Meanwhile, the innovative sentence-level model that he developed was trained specifically on single-sentence cases in the training data. For prediction on test data, he split the cases into sentences, evaluated each separately, and took only the highest score for sentences within the case. This was to accommodate the expectation that in natural language, speakers will frequently confine insulting comments to a single part of their speech.

Tuzzeg's model created over 100 feature groups (where a stem-based bigram is an example feature group—a group in the sense that the bigram process creates a vector of features), with the most important ones (ranked by impact) being the following:

```
stem subsequence based        0.66
stem based (unigrams, bigrams) 0.18
char ngrams based (sentence)   0.07
char ngrams based             0.04
all syntax                    0.006
all language models           0.004
all mixed                     0.002
```

This is interesting, in that it suggests that a set of feature translations that we aren't currently using is important in generating a usable solution. Particularly, the subsequence-based features are only a short step from our initial feature set, making it straightforward to add the extra feature:
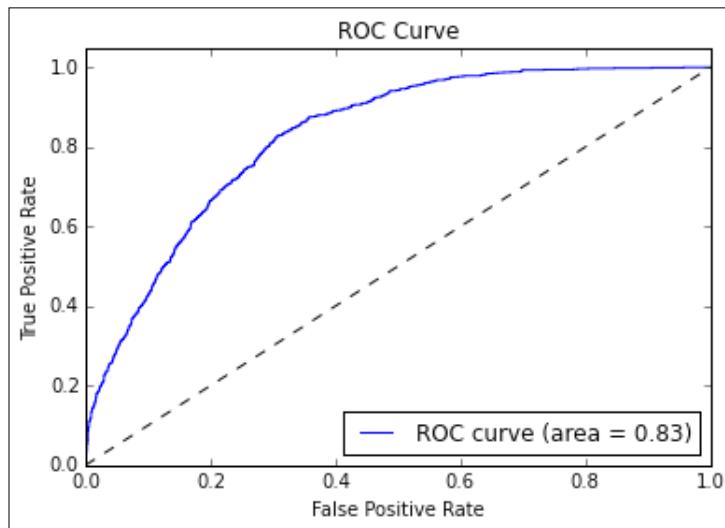
```
def subseq2(n, xs):
  l = len(xs)
  return ['%s %s' % (xs[i], xs[j]) for i in xrange(l-1) for j in
xrange(i+1, i+n+1) if j < l]

def getSubseq2(seqF, n):
  def f(row):
    seq = seqF(row)
    return set(seq + subseq2(n, seq))
  return f

Subseq2test = getSubseq2(line, 2)
```

This approach yields excellent results. While I'd encourage you to export Tuzzeg's own solution and apply it, you can also look at the `score_trolls_withsubseq.py` script provided in this project's repository to get a feeling for how powerful additional features can be incorporated.

With these additional features added, we see a dramatic improvement in our AUC score:

Running this code provides a very healthy `0.834` AUC score. This simply goes to show the power of thoughtful and innovative feature engineering; while the specific features generated in this chapter will serve you well in other contexts, specific hypotheses (such as hostile comments being isolated to specific sentences within a multi-sentence comment) can lead to very effective features.

As we've had the luxury of checking our reasoning against test data throughout this chapter, we can't reasonably say that we've worked under life-like conditions. We didn't take advantage of having access to the test data by reviewing it ourselves, but it's fair to say that knowing what the private leaderboard scored for this challenge made it easier for us to target the right fixes. In *Chapter 8*, *Ensemble Methods*, we'll be working on another tricky Kaggle problem in a more rigorous and realistic way. We'll also be discussing ensembles in depth!

Further reading

The quotes at the start of this chapter were sourced from the highly-readable Kaggle blog, No Free Hunch. Refer to `http://blog.kaggle.com/2014/08/01/learning-from-the-best/`.

There are many good resources for understanding NLP tasks. One fairly thorough, eight-part piece, is available online at `http://textminingonline.com/dive-into-nltk-part-i-getting-started-with-nltk`.

If you're keen to get started, one great option is to try Kaggle's for Knowledge NLP task, which is perfectly suited as a testbed for the techniques described in this chapter: `https://www.kaggle.com/c/word2vec-nlp-tutorial/details/part-1-for-beginners-bag-of-words`.

The Kaggle contest cited in this chapter is available at `https://www.kaggle.com/c/detecting-insults-in-social-commentary`.

For readers interested in further description of the ROC curve and the AUC measure, consider Tom Fawcett's excellent introduction, available at `https://ccrma.stanford.edu/workshops/mir2009/references/ROCintro.pdf`.

Summary

We've been introduced to a lot of useful and highly applicable skills in this chapter. In this chapter, we took a set of messy, complication-strewn text data and, through a series of rigorous steps, turned it into a large set of effective features. We began by picking up a set of data cleaning skills which stripped out a lot of the noise and problem elements, then we followed up by turning text into features using POS tagging and bag of words. In the process, you learned to apply a set of techniques that are widely applicable and very empowering, enabling us to solve difficult problems in many natural language processing contexts.

Through experimentation with multiple individual models and ensembles, we discovered that where a smarter algorithm might not yield a strong result, thorough and creative feature engineering can yield massive improvements in model performance.

7

Feature Engineering Part II

Introduction

We have recognized the importance of feature engineering. In the previous chapter, we discussed techniques that enable us to select from a range of features and work effectively to transform our original data into features, which can be effectively processed by the advanced ML algorithms that we have discussed thus far.

The adage *garbage in, garbage out* is relevant in this context. In earlier chapters, we have seen how image recognition and NLP tasks require carefully-prepared data. In this chapter, we will be looking at a more ubiquitous type of data: quantitative or categorical data that is collected from real-world applications.

Data of the type that we will be working with in this chapter is common to many contexts. We could be discussing telemetry data captured from sensors in a forest, game consoles, or financial transactions. We could be working with geological survey information or bioassay data collected through research. Regardless, the core principles and techniques remain the same.

In this chapter, you will be learning how to interrogate this data to weed out or mitigate quality issues, how to transform it into forms that are conducive to machine learning, and how to creatively enhance that data.

In general terms, the concepts that we'll be discussing in this chapter are as follows:

- The different approaches to feature set creation and the limits of feature engineering

- How to use a large set of techniques to enhance and improve an initial dataset

- How to tie in and use domain knowledge to understand valid options to transform and improve the clarity of existing data

- How we can test the value of individual features and feature combinations so that we only keep what we need

While we will begin with a detailed discussion of the underlying concepts, by the end of this chapter we will be working with multiple, iterative trials and using specialized tests to understand how helpful the features that we are creating will be to us.

Creating a feature set

The most important factor involved in successful machine learning is the quality of your input data. A good model with misleading, inappropriately normalized, or uninformative data will not see the same level of success anywhere near a model run over appropriately prepared data.

In some cases, you have the ability to specify data collection or have access to a useful, sizeable, and varied set of source data. With the right knowledge and skillset, you can use this data to create highly useful feature sets.

In general, having a strong knowledge as to how to construct good feature sets is very helpful as it enables you to audit and assess any new dataset for missed opportunities. In this chapter, we will introduce a design process and technique set that make it easier to create effective feature sets.

As such, we'll begin by discussing some techniques that we can use to extend or reinterpret existing features, potentially creating a large number of useful parameters to include in our models.

However, as we will see, there are limitations on the effective use of feature engineering techniques and we need to be mindful of the risks around engineered datasets.

Engineering features for ML applications

We have discussed what you can do about patching up data quality issues in your data and we have talked about how you can creatively use dimensions in what you have to join to external data.
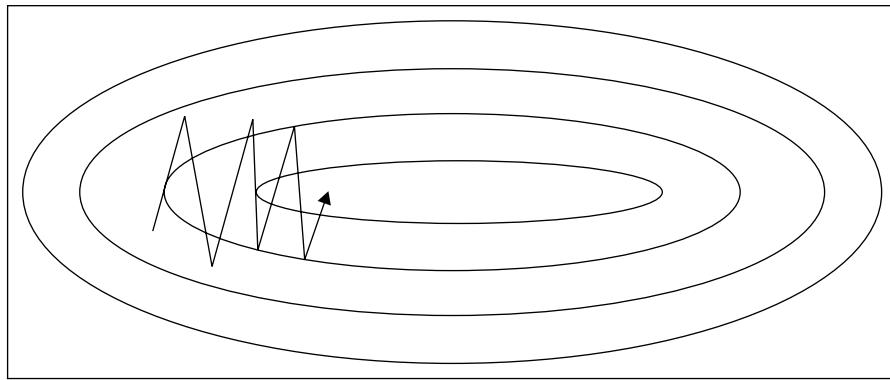
Once you have a reasonably well-understood and quality-checked set of data in front of you, there is usually still a significant amount of work needed before you can produce effective models from that data.

Using rescaling techniques to improve the learnability of features

The main challenge with directly feeding unprepared data to many machine learning models is that the algorithm is sensitive to the relative size of different variables. If your dataset has multiple parameters whose ranges differ, some algorithms will treat the variables whose variance is greater as indicative of more significant change than algorithms with smaller values and less variance.
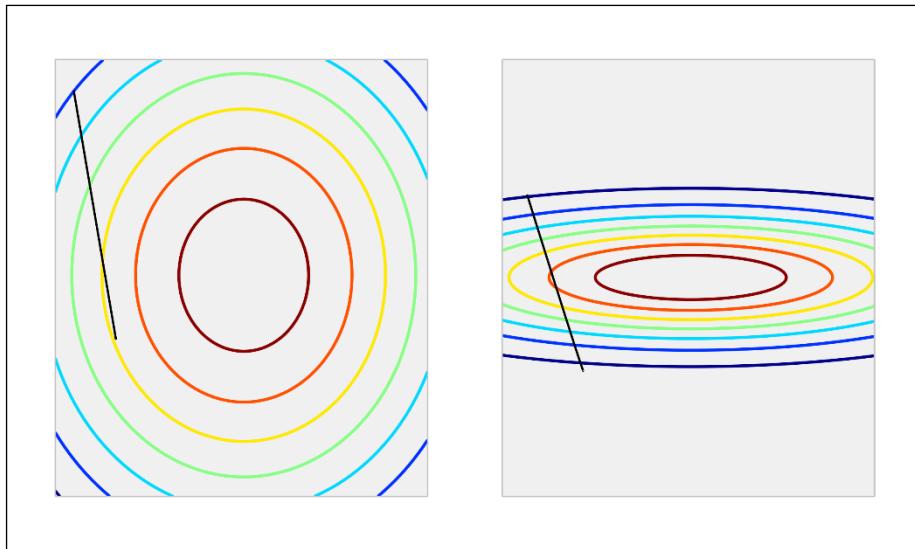
The key to resolving this potential problem is rescaling, a process by which parameter values' relative size is adjusted while retaining the initial ordering of values within each parameter (a monotonic translation).

Gradient descent algorithms (which include most deep learning algorithms — `http://sebastianruder.com/optimizing-gradient-descent/`) are significantly more efficient if the input data is scaled prior to training. To understand why, we'll resort to drawing some pictures. A given series of training steps may appear as follows:

When applied to unscaled data, these training steps may not converge effectively (as per the left-hand example in the following diagram).

With each parameter having a differing scale, the parameter space in which models are attempting to train can be highly distorted and complex. The more complex this space, the harder it becomes to train a model within it. This is an involved subject that can be effectively described, in general terms, through a metaphor, but for readers looking for a fuller explanation there is an excellent reference in this chapter's *Further reading* section. For now, it is not unreasonable to think in terms of gradient descent models during training as behaving like marbles rolling down a slope. These marbles are prone to *getting stuck* in saddle points or other complex geometries on the slope (which, in this context, is the surface created by our model's objective function—the learning function whose output our models typically train to minimize). With scaled data, however, the surface becomes more regularly-shaped and training can become much more effective:

The classic example is a linear rescaling between *0* and *1*; with this method, the largest parameter value is rescaled to *1*, the smallest to *0*, with intermediate values falling in the *0-1* interval, proportionate to their original size relative to the largest and smallest values. Under such a transformation, the vector *[0,10,25,20,18]*, for instance, would become *[0,0.4, 1, 0.8, 0.72]*.

The particular value of this transformation is that, for multiple data points that may vary in magnitude in its raw form, the rescaled features will sit within the same range, enabling your machine learning algorithm to train on meaningful information content.

This is the most straightforward rescaling option, but there are some nonlinear scaling alternatives that can be much more helpful in the right circumstances; these include square scaling, square root scaling, and perhaps most commonly, log-scaling.

Log-scaling of parameter values is very common in physics and in contexts where the underlying data is frequently affected by a power law (for example, an exponential growth in y given a linear increase in x).

Unlike linear rescaling, log-scaling adjusts the relative spacing between data cases. This can be a double-edged sword. On the one hand, log-scaling handles outlying cases very well. Let's take an example dataset describing individual net wealth for members of a fictional population, described by the following summary statistics:

Statistic	Wealth
Min	1
First Quartile	42.5
Mean	3205433.343
Median	600
Third Quartile	1358
Max	10000000000

Prior to rescaling, this population is hugely skewed toward that single individual with absurd net worth. The distribution of cases per decile is as follows:

Range	Count of Cases
0 > 0.1	3060
0.1 > 0.2	0
0.2 > 0.3	0
0.3 > 0.4	0
0.4 > 0.5	0
0.5 > 0.6	0
0.6 > 0.7	0
0.7 > 0.8	0
0.8 > 0.9	0
0.9 > 1	1

After log-scaling, this distribution is far friendlier:

Range	Count of Cases
0 > 0.1	740
0.1 > 0.2	1633
0.2 > 0.3	544
0.3 > 0.4	141
0.4 > 0.5	0
0.5 > 0.6	1
0.6 > 0.7	0
0.7 > 0.8	1
0.8 > 0.9	0
0.9 > 1	1

We could've chosen to take scaling further and drawn out the first half of this distribution more by doing that. In this case, log-10 normalization significantly reduces the impact of these outlying values, enabling us to retain outliers in the dataset without losing detail at the lower end.

With this said, it's important to note that in some contexts, that same enhancement of clustered cases can enhance noise in variant parameter values and create the false impression of greater spacing between values. This tends not to negatively affect how log-scaling handles outliers; the impact is usually seen for groups of smaller-valued cases whose original values are very similar.

The challenges created by introducing nonlinearities through log-scaling are significant and in general, nonlinear scaling is only recommended for variables that you understand and have a nonlinear relationship or trend underlying them.

Creating effective derived variables

Rescaling is a standard part of preprocessing in many machine learning applications (for instance, almost all neural networks). In addition to rescaling, there are other preparatory techniques, which can improve model performance by strategically reducing the number of parameters input to the model. The most common example is of a derived measure that takes multiple existing data points and represents them within a single measure.

These are extremely prevalent; examples include acceleration (as a function of velocity values from two points in time), body mass index (as a function of height, weight, and age), and **price-earnings** (**P/E**) ratio for stock scoring. Essentially, any derived score, ratio, or complex measure that you ever encounter is a combination score formed from multiple components.

For datasets in familiar contexts, many of these pre-existing measures will be well-known. Even in relatively well-known areas, however, looking for new supporting measures or transformations using a mix of domain knowledge and existing data can be very effective. When thinking through derived measure options, some useful concepts are as follows:

- **Two variable combinations**: Multiplication, division, or normalization of the *n* parameter as a function of the *m* parameter.

- **Measures of change over time**: A classic example here is acceleration or 7D change in a measure. In more complex contexts, the slope of an underlying time series function can be a helpful parameter to work with instead of working directly with the current and past values.

- **Subtraction of a baseline**: Using a base expectation (a flat expectation such as the *baseline churn rate*) to recast a parameter in terms of that baseline can be a more immediately informative way of looking at the same variable. For the churn example, we could generate a parameter that describes churn in terms of deviation from an expectation. Similarly, in stock trading cases, we might look at closing price in terms of the opening price.

- **Normalization**: Following on from the previous case, normalization of parameter values based on the values of another parameter or baseline that is dynamically calculated given properties of other variables. One example here is *failed transaction rate*; in addition to looking at this value as a raw (or rescaled) count, it often makes sense to normalize this in terms of attempted transactions.

Creative recombination of these different elements lets us build very effective scores. Sometimes, for instance, a parameter that tells us the slope of customer engagement (declining or increasing) over time needs to be conditioned on whether that customer was previously highly engaged or hardly engaged, as a slight decline in engagement might mean very different things in each context. It is the data scientist's job to effectively and creatively feature sets that capture these subtleties for a given domain.

So far, this discussion has focused on numerical data. Often, however, useful data is locked up inside non-numeric parameters such as codes or categorical data. Accordingly, we will next discuss a set of effective techniques to turn non-numeric features into usable parameters.

Reinterpreting non-numeric features

A common challenge, which can be problematic and problem-specific, is how non-numeric features are treated. Frequently, valuable information is encoded within non-numerical shorthand values. In the case of stock trades, for instance, the identity of the stock itself (for example, AAPL) as well as that of the buyer and seller is interesting information that we expect to relate meaningfully to our problem. Taking this example further, we might also expect some stocks to trade differently from others even within the industry, and organizational differences within companies, which may occur at some or all points of time, also provide important context.

One simple option that works in some cases is building an aggregation or series of aggregations. The most obvious example is a count of occurrences with the possibility of creating extended measures (changes in count between two time windows) as described in the preceding section.

Building summary statistics and reducing the number of rows in the dataset introduces the risk of reducing the amount of information that your model has available to learn from (increasing the risk of model fragility and overfitting). As such, it's generally a bad idea to extensively aggregate and reduce input data. This is doubly true with deep learning techniques, such as the algorithms discussed and used in Chapters 2-4.

Rather than extensively using aggregation-based approaches, let's look at an alternative way of translating string-encoded values into numerical data. Another very popular class of techniques is encoding, with the most common encoding tactic being one-hot encoding. One-hot encoding is the process of turning a series of categorical responses (for example, age groups) into a set of binary variables, with each response option (for example, 18-30) represented by its own binary variable. This is a little more intuitive when presented visually:

Case	Age	Gender
1	22	M
2	25	M
3	34	F
4	23	M
5	25	F
6	41	F

After encoding, this dataset of categorical and continuous variables becomes a tensor of binary variables:

Case	Age_22	Age_23	Age_25	Age_34	Age_41	Gender_F	Gender_M
1	1	0	0	0	0	0	1
2	0	0	1	0	0	0	1
3	0	0	0	1	0	1	0
4	0	1	0	0	0	0	1
5	0	0	0	0	0	1	0
6	0	0	0	0	1	1	0

The advantage that this presents is significant; it enables us to tap into the very valuable tag information contained within a lot of datasets without aggregation or risk of reducing the information content of the data. Furthermore, one-hot allows us to separate specific response codes for encoded variables into separate features, meaning that we can identify more or less meaningful codes for a specific variable and only retain the important values.
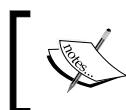
Another very effective technique, used primarily for text codes, is known as the **hash trick**. A hash, in simple terms, is a function that translates data into a numeric representation. Hashes will be a familiar concept to many, as they're frequently used to encode sensitive parameters and summarize otherwise bulky data. In order to get the most out of the hash trick, however, it's important to understand how the trick works and what can be done with it.

We can use hashing to turn a text phrase into a numeric value that we can use as an identifier for that phrase. While there are many applications of different hashing algorithms, in this context even a simple hash makes it straightforward to turn string keys and codes into numerical parameters that we can model effectively.

A very simple hash might turn each alphabet character into a corresponding number. *a* would become *1*, *b* would be *2*, and so on. Hashes could be generated for words and phrases by summing those values. The phrase *cat gifs* would translate under this scheme as follows:

```
Cat: 3 + 1 + 20
Gifs: 7 + 9 + 6 + 19
Total: 65
```

This is a terrible hash for two reasons (quite disregarding the fact that the input contains junk words!). Firstly, there's no real limit on how many outputs it can present. When one remembers that the whole point of the hash trick is to provide dimensionality reduction, it stands to reason that the number of possible outputs from a hash must be bounded! Most hashes limit the range of numbers that they output, so part of the decision in terms of selecting a hash is related to the number of features you'd prefer your model to have.

> One common behavior is to choose a power of two as the hash range; this tends to speed things up by allowing bitwise operations during the hashing process.

The other reason that this hash kind of sucks is that changes to the word have a small impact rather than a large one. If *cat* became *bat*, we'd want our hash output to change substantially. Instead, it changes by one (becoming *64*). In general, a good hash function is one where a small change in the input text will cause a large change in the output. This is partly because language structures tend to be very uniform (thus scoring similarly), but slightly different sets of nouns and verbs within a given structure tend to confer very different meanings to one another (*the cat sat on the mat* versus *the car sat on the cat*).

So we've described hashing. The hash trick takes things a little further. Hypothetically, turning every word into a hashed numerical code is going to lead to a large number of *hash collisions*—cases where two words have the same hash value. Naturally, these are rather bad.

Handily, there's a distribution underlying how frequently different terms are used that work in our favor. Called the **Zipf distribution**, it entails that the probability of encountering the nth most common term is approximated by *P(n) = 0.1/n* up to around 1,000 (Zipf's law). This entails that each term is much less likely to be encountered than the preceding term. After *n = 1000*, terms tend to be sufficiently obscure that it's unlikely to encounter two that have the same hash in one dataset.

At the same time, a good hashing function has a limited range and is significantly affected by small changes in input. These properties make the hash collision chance largely independent of term usage frequency.

These two concepts—Zipf's law and a good hash's independence of hash collision chance and term usage frequency—mean that there is very little chance of a hash collision, and where one occurs it is overwhelmingly likely to be between two infrequently-used words.

This gives the hash trick a peculiar property. Namely, it is possible to reduce the dimensionality of a set of text input data massively (from tens of thousands of naturally occurring words to a few hundred or fewer) without reducing the performance of a model trained on hashed data, compared to training on unhashed bag-of-words features.

Proper use of the hash trick enables a lot of possibilities, including augmentations to the techniques that we discussed (specifically, bag-of-words). References to different hashing implementations are included in the *Further reading* section at the end of this chapter.

Using feature selection techniques

Now that we have a good selection of options for feature creation, as well as an understanding of the creative feature engineering possibilities, we can begin building our existing features into more effective variants. Given this new-found feature engineering skillset, we run the risk of creating extensive and hard-to-manage datasets.

Adding features without limit increases the risk of model fragility and overfitting for certain types of models. This is tied to the complexity of the trends that you're attempting to model. In the simplest case, if you're attempting to identify a significant distinction between two large groups, your model is likely to support a large number of features. However, as the model you need to fit to make this distinction becomes more complex and as the group sizes that you have to work with become smaller, adding more and more features can harm the model's ability to classify consistently and effectively.

This challenge is compounded by the fact that it isn't always obvious which parameter or variation is best-suited for the task. Suitability can vary by the underlying model; decision forests, for instance, don't perform any better with monotonic transformations (that is, transformations that retain the initial ordering of data cases; one example is log-scaling) than with the unscaled base data; however, for other algorithms, the choice to rescale and the rescaling method used are both very impactful choices.

Traditionally, the quantity of features and limits on the parameter amount were tied to the desire to develop a mathematical function that relates key inputs to the desired outcome scores. In this context, additional parameters needed to be incorporated as moving or nuisance variables.

Each new parameter introduces another dimension that makes the modeled relationship more complex and the resultant model more likely to be overfitting the data that exists. A trivial example is if you introduce a parameter that is just a unique label for each case; at this point, your algorithm will just learn those labels, making it very likely that your model fails entirely when introduced to a new dataset.

Less trivial examples are no less problematic; the proportion of cases to features becomes very important when your features are separating cases down to very small groups. In short, increasing the complexity of the modeled function causes your model to be more liable to overfit and adding features can exacerbate this effect. According to this principle, we should be beginning with very small datasets and adding parameters only after justifying that they improve the model.

However, in recent times, an opposing methodology—now generally seen as being part of a common way of *doing data science*—has gained ground. This methodology suggests that it's a good idea to add very large feature sets to incorporate every potentially valuable feature and *work down* to a smaller feature set that does the job.

This methodology is supported by techniques that enable decisions to be made over huge feature sets (potentially hundreds or thousands of features) and that tend to operate in a *brute force* manner. These techniques will exhaustively test feature combinations, running models in series or in parallel until the most effective parameter subsets are identified.

These techniques work, which is why this methodology has become popular. It is definitely worth knowing about these techniques, if not using them, so you'll be learning how to apply them later in this chapter.

The main disadvantage around using brute force techniques for feature selection is that it becomes very easy to trust the outcomes of the algorithm, irrespective of what the features it selects actually mean. It is sensible to balance the use of highly effective, black-box algorithms against domain knowledge and an understanding of what's being undertaken. Therefore, this chapter will enable you to use techniques from both paradigms (*build up* and *build down*) so that you can adapt to different contexts. We'll begin by learning how to narrow down the feature set that you have to work with, from many features to the most valuable subset.

Performing feature selection

Having built a large dataset, often the next challenge one faces is how to narrow down the options to retain only the most effective data. In this section, we'll discuss a variety of techniques that support feature selection, working by themselves or as wrappers to familiar algorithms.

These techniques include correlation analysis, regularization techniques, and **Recursive Feature Elimination** (**RFE**). When we're done, you'll be able to confidently use these techniques to support your selection of feature sets, potentially saving yourself a significant amount of work every time you work with a new dataset!

Correlation

We'll begin our discussion of feature selection by looking for a simple source of major problems for regression models: multicollinearity. Multicollinearity is the fancy name for moderate or high degrees of correlation between features in a dataset. An obvious example is how pizza slice count is collinear with pizza price.

There are two types of multicollinearity: structural and data-based. Structural multicollinearity occurs when the creation of new features, such as feature *f1* from feature *f*, creates multiple features that may be highly correlated with one another. Data-based multicollinearity tends to occur when two variables are affected by the same causative factor.

Both kinds of multicollinearity can cause some unfortunate effects. In particular, our models' performance tends to become affected by which feature combinations are used; when collinear features are used, the performance of our model will tend to degrade.

In either case, our approach is simple: we can test for multicollinearity and remove underperforming features. Naturally, underperforming features are ones that add very little to model performance. They might be underperforming because they replicate information available in other features, or they may simply not provide data that is meaningful to the problem at hand. There are multiple ways to test for weak features as many feature selection techniques will sift out multicollinear feature combinations and recommend their removal if they're underperformant.

In addition, there is a specific multicollinearity test that's worth considering; namely, inspecting the eigenvalues of our data's correlation matrix. Eigenvectors and eigenvalues are fundamental concepts in the matrix theory with many prominent applications. More details are given at the end of this chapter. For now, suffice it to say that eigenvalues in the correlation matrix generated by our dataset provide us with a quantified measure of multicollinearity. Consider a set of eigenvalues as indicative of how much "new information content" our features bring to the dataset; a low eigenvalue suggests that the data may be correlated with other features. For an example of this at work, consider the following code, which creates a feature set and then adds collinearity to features *0, 2,* and *4*:

```
import numpy as np

x = np.random.randn(100, 5)
noise = np.random.randn(100)
x[:,4] = 2 * x[:,0] + 3 * x[:,2] + .5 * noise
```

When we generate the correlation matrix and compute eigenvalues, we find the following:

```
corr = np.corrcoef(x, rowvar=0)
w, v = np.linalg.eig(corr)

print('eigenvalues of features in the dataset x')
print(w)

eigenvalues of features in the dataset x
[ 0.00716428  1.94474029  1.30385565  0.74699492  0.99724486]
```

Clearly, our *0th* feature is suspect! We can then inspect the eigenvalues of this feature via calling v:

```
print('eigenvalues of eigenvector 0')
print(v[:,0])

eigenvalues of eigenvector 0
[-0.35663659 -0.00853105 -0.62463305  0.00959048  0.69460718]
```

From the small values of features in position one and three, we can tell that features *2* and *4* are highly multicollinear with feature *0*. We ought to remove two of these three features before proceeding!

LASSO

Regularized methods are among the most helpful feature selection techniques as they provide sparse solutions: ones where weaker features return zero, leaving only a subset of features with real coefficient values.

The two most used regularization models are L1 and L2 regularization, referred to as LASSO and ridge regression respectively in linear regression contexts.

Regularized methods function by adding a penalty to the loss function. Instead of minimizing a loss function $E(X,Y)$, the penalty leads to $E(X,Y) + a||w||$. The hyperparameter a relates to the amount of regularization (enabling us to tune the strength of our regularization and thus the proportion of the original feature set that is selected).

In LASSO regularization, the specific penalty function used is $a\sum n i=1|wi|$. Each non-zero coefficient adds to the size of the penalty term, forcing weaker features to return coefficients of *0*. Selecting an appropriate penalty term can be achieved using scikit-learn's parameter optimization support for hyperparameters. In this case, we'll be using `estimator.get_params()` to perform a grid search for appropriate hyperparameter values. For more information on how grid searches operate, see the *Further reading* section at the end of this chapter.

In scikit-learn, logistic regression is provided with an L1 penalty for classification. Meanwhile, the LASSO module is provided for linear regression. For now, let's begin by applying LASSO to an example dataset. In this case, we'll use the Boston housing dataset:

```
fromsklearn.linear_model import Lasso
fromsklearn.preprocessing import StandardScaler
fromsklearn.datasets import load_boston

boston = load_boston()
scaler = StandardScaler()
X = scaler.fit_transform(boston["data"])
Y = boston["target"]
names = boston["feature_names"]

lasso = Lasso(alpha=.3)
lasso.fit(X, Y)

print "Lasso model: ", pretty_print_linear(lasso.coef_, names, sort =
True)

Lasso model: -3.707 * LSTAT + 2.992 * RM + -1.757 * PTRATIO + -1.081
* DIS + -0.7 * NOX + 0.631 * B + 0.54 * CHAS + -0.236 * CRIM + 0.081 *
ZN + -0.0 * INDUS + -0.0 * AGE + 0.0 * RAD + -0.0 * TAX
```

Several of the features in the original set returned a correlation of `0.0`. Increasing the correlation makes the solution increasingly sparse. For instance, we see the following results when `alpha = 0.4`:

```
Lasso model: -3.707 * LSTAT + 2.992 * RM + -1.757 * PTRATIO + -1.081
* DIS + -0.7 * NOX + 0.631 * B + 0.54 * CHAS + -0.236 * CRIM + 0.081 *
ZN + -0.0 * INDUS + -0.0 * AGE + 0.0 * RAD + -0.0 * TAX
```

We can immediately see the value of L1 regularization as a feature selection technique. However, it is important to note that L1 regularized regression is unstable. Coefficients can vary significantly, even with small data changes, when features in the data are correlated.

This problem is effectively addressed with L2 regularization, or ridge regression, which develops a feature coefficient with different applications. L2 normalization adds an additional penalty, the L2 norm penalty, to the loss function. This penalty takes the form ($a\sum_{i=1}^{n}w_i^2$). A sharp-eyed reader will notice that, unlike the L1 penalty ($a\sum_{i=1}^{n}|w_i|$), the L2 penalty uses squared coefficients. This causes the coefficient values to be spread out more evenly and has the added effect that correlated features tend to receive similar coefficient values. This significantly improves stability as the coefficients no longer fluctuate on small data changes.

However, L2 normalization isn't as directly useful for feature selection as L1. Rather, as interesting features (with predictive power) tend to have non-zero coefficients, L2 is more useful as an exploratory tool allowing inference about the quality of features in the classification. It has the added merit of being more stable and reliable than L1 regularization.

Recursive Feature Elimination

RFE is a greedy, iterative process that functions as a wrapper over another model, such as an SVM (SVM-RFE), which it repeatedly runs over different subsets of the input data.

As with LASSO and ridge regression, our goal is to find the best-performing feature subset. As the name suggests, on each iteration a feature is set aside allowing the process to be repeated with the rest of the feature set until all features in the dataset have been eliminated. The ordering with which features are eliminated becomes their rank. After multiple iterations with incrementally smaller subsets, each feature is accurately scored and relevant subsets can be selected for use.

To get a better understanding of how this works, let's look at a simple example. We'll use the (by now familiar) digits dataset to understand how this approach works in practice:

```
print(__doc__)

from sklearn.svm import SVC
fromsklearn.datasets import load_digits
fromsklearn.feature_selection import RFE
importmatplotlib.pyplot as plt

digits = load_digits()
X = digits.images.reshape((len(digits.images), -1))
y = digits.target
```
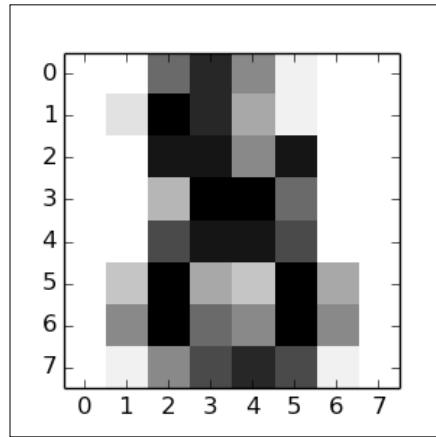
We'll use an SVM as our base estimator via the `SVC` operator for **Support Vector Classification** (**SVC**). We then apply the RFE wrapper over this model. RFE takes several arguments, with the first being a reference to the estimator of choice. The second argument is `n_features_to_select`, which is fairly self-explanatory. In cases where the feature set contains many interrelated features whose subsets possess multivariate distributions that are highly effective classification features, it's possible to opt for feature combinations of two or more.

Stepping enables the removal of multiple features on each iteration. When given a value between *0.0* and *1.0*, each step enables the removal of a percentage of the feature set, corresponding to the proportion given in the step argument:
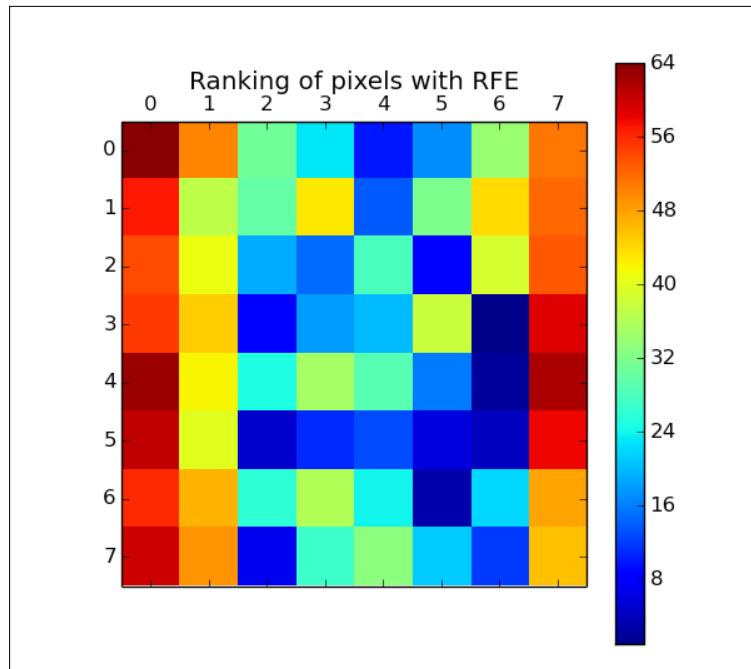
```
svc = SVC(kernel="linear", C=1)
rfe = RFE(estimator=svc, n_features_to_select=1, step=1)
rfe.fit(X, y)
ranking = rfe.ranking_.reshape(digits.images[0].shape)

plt.matshow(ranking)
plt.colorbar()
plt.title("Ranking of pixels with RFE")
plt.show()
```

Given that we're familiar with the digits dataset, we know that each instance is an 8 x 8 image of a handwritten digit, as shown in the following image. Each image is located in the center of the 8 x 8 grid:

When we apply RFE over the digits dataset, we can see that it broadly captures this information in applying a ranking:

The first pixels to be cut were in and around the (typically empty) vertical edges of the image. Next, the algorithm began culling normally whitespace areas around the vertical edges or near the top of the image. The pixels that were retained longest were those that enabled the most differentiation between the different characters—pixels that would be present for some numbers and not for others.

This example gives us great visual confirmation that RFE works. What it doesn't give us is evidence for how consistently the technique works. The stability of RFE is dependent on the stability of the base model and, in some cases, ridge regression will provide a more stable solution. (For more information on which cases and the conditions involved, consult the *Further reading* section at the end of this chapter.)

Genetic models

Earlier in this chapter, we discussed the existence of algorithms that enable feature selection with very large parameter sets. Some of the most prominent techniques of this type are genetic algorithms, which emulate natural selection to generate increasingly effective models.

A genetic solution for feature selection works roughly as follows:

- An initial set of variables (predictors is the term typically used in this context) are combined into multiple subsets (candidates) and a performance measure is calculated for each candidate
- The predictors from candidates with the best performance are randomly recombined into a new iteration (generation) of models
- During this recombination step, for each subset there is the probability of a mutation, whereby a predictor may be added or removed from a subset

This algorithm typically iterates for multiple generations. The appropriate iteration amount is dependent on the complexity of the dataset and the model required. As with gradient descent techniques, the typical relationship between the performance and iteration count is present for genetic algorithms, where performance improvement declines nonlinearly as the count of iterations increases, eventually hitting a minimum before the overfitting risk increases.

To find an effective iteration count, we can perform testing using training data; by running the model for a large number of iterations and plotting the **Root Mean Squared Error** (**RMSE**), we're able to find an appropriate amount of iterations given our input data and model configuration.

Let's talk in a little more detail about what happens within each generation. Specifically, let's talk about how candidates are created, how performance is scored, and how recombination is performed.

The candidates are initially configured to use a random sample of the available predictors. There is no hard and fast rule concerning how many predictors to use in the first generation; it depends on how many features are available, but it's common to see first generation candidates using 50% to 80% of the available features (with a smaller percentage used in cases with more features).

The fitness measure can be difficult to define, but a common practice is to use two forms of cross-validation. Internal cross-validation (testing each model solely in the context of its own parameters without comparing models) is typically used to track performance at a given iteration; the fitness measures from internal cross-validation are used to select models to recombine in the next generation. External cross-validation (testing against a dataset that was not used in validation at any iteration) is also needed in order to confirm that the search process produced a model that has not overfitted to the internal training data.

Recombination is controlled by three key parameters: mutation, cross-over probabilities, and elitism. The latter is an optional parameter that one may use to reserve n-many of the top-performing models from the current generation; by doing so, one may preserve particularly effective candidates from being lost entirely during recombination. This can be done while also using that candidate in mutated variants and/or using them as parents to next-generation candidates.

The mutation probability defines the chance of a next-generation model being randomly readjusted (via some predictors, typically one, being added or removed). Mutation tends to help the genetic algorithm maintain a broad coverage of the candidate variables, reducing the risk of falling into a parameter-local solution.

Cross-over probability defines the likelihood that a pair of candidates will be selected for recombination into a next-generation model. There are several cross-over algorithms: parts of each parent's feature set might be spliced (for example, first half/second half) into the child or a random selection of each parent's features might be used. Common features to both parents might also be used by default. Random sampling from the set of both parent's unique predictors is a common default approach.

These are the main parts of a general genetic algorithm, which can be used as a wrapper to existing models (logistic regression, SVM, and others). The technique described here can be varied in many different ways and is related to feature selection techniques used slightly differently across multiple quantitative fields. Let's take the theory that we've covered thus far and start applying it to a practical example.

Feature engineering in practice

Depending on the modeling technique that you're using, some of this work may be more valuable than other parts. Deep learning algorithms tend to perform better on less-engineered data than shallower models and it might be that less work is needed to improve results.

The key to understanding what is needed is to iterate quickly through the whole process from dataset acquisition to modeling. On a first pass with a clear target for model accuracy, find the acceptable minimum amount of processing and perform that. Learn whatever you can about the results and make a plan for the next iteration.

To show how this looks in practice, we'll work with an unfamiliar, high-dimensional dataset, using an iterative process to generate increasingly effective modeling.

I was recently living in Vancouver. While it has many positive qualities, one of the worst things about living in the city was the somewhat unpredictable commute. Whether I was traveling by car or taking Translink's Skytrain system (a monorail-meets-rollercoaster high-speed line), I found myself subject to hard-to-predict delays and congestion issues.

In the spirit of putting our new feature engineering skillset into practice, let's take a look at whether we can improve this experience by taking the following steps:

- Writing code to harvest data from multiple APIs, including text and climate streams
- Using our feature engineering techniques to derive variables from this initial data
- Testing our feature set by generating commute delay risk scores

Unusually, in this example, we'll focus less on building and scoring a highly performant model. Instead, our focus is on creating a self-sufficient solution that you can adjust and apply for your own local area. While it suits the goals of the current chapter to take this approach, there are two additional and important motivations.

Firstly, there are some challenges around sharing and making use of Twitter data. Part of the terms of use of Twitter's API is an obligation on the developer to ensure that any adjustments to the state of a timeline or dataset (including, for instance, the deletion of a tweet) are reproduced in datasets that are extracted from Twitter and publicly shared. This makes the inclusion of real Twitter data in this chapter's GitHub repository impractical. Ultimately, this makes it difficult to provide reproducible results from any downstream model based on streamed data because users will need to build their own stream and accumulate data points and because variations in context (such as seasonal variations) are likely to affect model performance.

The second element here is simple enough: not everybody lives in Vancouver! In order to generate something of value to an end user, we should think in terms of an adjustable, general solution rather than a geographically-specific one.

The code presented in the next section is therefore intended to be something to build from and develop. It offers potential as the basis of a successful commercial app or simply a useful, data-driven life hack. With this in mind, review this chapter's content (and leverage the code in the associated code directory) with an eye to finding and creating new applications that fit your own situation, locally available data, and personal needs.

Acquiring data via RESTful APIs

In order to begin, we're going to need to collect some data! We're going to need to look for rich, timestamped data that is captured at sufficient frequency (preferably at least one record per commute period) to enable model training.

A natural place to begin with is the Twitter API, which allows us to harvest recent tweet data. We can put this API to two uses.

Firstly, we can harvest tweets from official transit authorities (specifically, bus and train companies). These companies provide transit service information on delays and service disruptions that, helpfully for us, takes a consistent format conducive to tagging efforts.

Secondly, we can tap into commuter sentiment by listening for tweets from the geographical area of interest, using a customized dictionary to listen for terms related to cases of disruption or the causes thereof.

In addition to mining the Twitter API for data to support our model, we can leverage other APIs to extract a wealth of information. One particularly valuable source of data is the **Bing Traffic API**. This API can be easily called to provide traffic congestion or disruption incidents across a user-specified geographical area.

In addition, we can leverage weather data from the **Yahoo Weather API**. This API provides the current weather for a given location, taking zip codes or location input. It provides a wealth of local climate information including, but not limited to, temperature, wind speed, humidity, atmospheric pressure, and visibility. Additionally, it provides a text string description of current conditions as well as forecast information.

While there are other data sources that we can consider tying into our analysis, we'll begin with this data and see how we do.

Testing the performance of our model

In order to meaningfully assess our commute disruption prediction attempt, we should try to define test criteria and an appropriate performance score.

What we're attempting to do is identify the risk of commute disruption on the current day, each day. Preferably, we'd like to know the commute risk with sufficient advance notice that we can take action to mitigate it (for example, by leaving home earlier).

In order to do this, we're going to need three things:

- An understanding of what our model is going to output
- A measure we can use to quantify model performance
- Some target data we can use to score model performance according to our measure

We can have an interesting discussion about why this matters. It can be argued, effectively, that some models are information in purpose. Our commute risk score, it might be said, is useful insofar as it generates information that we didn't previously have.

The reality of the situation, however, is that there is inalienably going to be a performance criterion. In this case, it might simply be my satisfaction with the results output by my model, but it's important to be aware that there is always some performance criterion at play. Quantifying performance is therefore valuable, even in contexts where a model appears to be informational (or even better, unsupervised). This makes it prudent to resist the temptation to waive performance testing; at least this way, you have a quantified performance measure to iteratively improve on.

A sensible starting point is to assert that our model is intended to output a numerical score in a *0-1* range for outbound (home to work) commutes on a given day. We have a few options with regard to how we present this score; perhaps the most obvious option would be to apply a log rescaling to the data. There are good reasons to log-scale and in this situation it might not be a bad idea. (It's not unlikely that the distribution of commute delay time obeys a power law.) For now, we won't reshape this set of scores. Instead, we'll wait to review the output of our model.

In terms of delivering practical guidance, a *0-1* score isn't necessarily very helpful. We might find ourselves wanting to use a bucketed system (such as high risk, mid risk, or low risk) with bucket boundaries at specific boundaries in the *0-1* range. In short, we would transition to treating the problem as a multiclass classification problem with categorical output (class labels), rather than as a regression problem with a continuous output.

This might improve model performance. (More specifically, because it'll increase the margin of free error to the full breadth of the relevant bucket, which is a very generous performance measure.) Equally though, it probably isn't a great idea to introduce this change on the first iteration. Until we've reviewed the distribution of real commute delays, we won't know where to draw the boundaries between classes!

Next, we need to consider how we measure the performance of our model. The selection of an appropriate scoring measure generally depends on the characteristics of the problem. We're presented with a lot of options around classifier performance scoring. (For more information around performance measures for machine learning algorithms, see the *Further reading* section at the end of this chapter.)

One way of deciding which performance measure is suitable for the task at hand is to consider the confusion matrix. A confusion matrix is a table of contingencies; in the context of statistical modeling, they typically describe the label prediction versus actual labels. It is common to output a confusion matrix (particularly for multiclass problems with more classes) for a trained model as it can yield valuable information about classification failures by failure type and class.

In this context, the reference to a confusion matrix is more illustrative. We can consider the following simplified matrix to assess whether there is any contingency that we don't care about:

		Actual Result	
		TRUE	FALSE
Prediction	TRUE	**True Positive**	**False Positive**
	FALSE	**False Negative**	**True Negative**

In this case, we care about all four contingency types. False negatives will cause us to be caught in unexpected delays, while false positives will cause us to leave for our commute earlier than necessary. This implies that we want a performance measure that values both high sensitivity (true positive rate) and high specificity (false positive rate). The ideal measure, given this, is **area under the curve** (**AUC**).

The second challenge is how to measure this score; we need some target against which to predict. Thankfully, this is quite easy to obtain. I do, after all, have a daily commute to do! I simply began self-recording my commute time using a stopwatch, a consistent start time, and a consistent route.

It's important to recognize the limitations of this approach. As a data source, I am subject to my own internal trends. I am, for instance, somewhat sluggish before my morning coffee. Similarly, my own consistent commute route may possess local trends that other routes do not. It would be far better to collect commute data from a number of people and a number of routes.

However, in some ways, I was happy with the use of this target data. Not least because I am attempting to classify disruption to my own commute route and would not want natural variance in my commute time to be misinterpreted through training, say, against targets set by some other group of commuters or routes. In addition, given the anticipated slight natural variability from day-to-day, should be disregarded by a functional model.

It's rather hard to tell what's good enough in terms of model performance. More precisely, it's not easy to know when this model is outperforming my own expectations. Unfortunately, not only do I not have any very reliable with regard to the accuracy of my own commute delay predictions, it also seems unlikely that one person's predictions are generalizable to other commutes in other locations. It seems ill-advised to train a model to exceed a fairly subjective target.

Let's instead attempt to outperform a fairly simple threshold—a model that naively suggests that every single day will not contain commute delays. This target has the rather pleasing property of mirroring our actual behavior (in that we tend to get up each day and act as though there will not be transit disruption).

Of the 85 target data cases, 14 commute delays were observed. Based on this target data and the score measure we created, our target to beat is therefore *0.5*.

Twitter

Given that we're focusing this example analysis on the city of Vancouver, we have an opportunity to tap into a second Twitter data source. Specifically, we can use service announcements from Vancouver's public transit authority, Translink.

Translink Twitter

As noted, this data is already well-structured and conducive both to text mining and subsequent analysis; by processing this data using the techniques we reviewed in the last two chapters, we can clean the text and then encode it into useful features.

We're going to apply the Twitter API to harvest Translink's tweets over an extended period. The Twitter API is a pretty friendly piece of kit that is easy enough to work with from Python. (For extended guidance around how to work with the Twitter API, see the *Further reading* section at the end of this chapter!) In this case, we want to extract the date and body text from the tweet. The body text contains almost everything we need to know, including the following:

- The nature of the tweet (delay or non-delay)
- The station affected
- Some information as to the nature of the delay

One element that adds a little complexity is that the same Translink account tweets service disruption information for Skytrain lines and bus routes. Fortunately, the account is generally very uniform in the terms that it uses to describe service issues for each service type and subject. In particular, the Twitter account uses specific hashtags (**#RiderAlert** for bus route information, **#SkyTrain** for train-related information, and **#TransitAlert** for general alerts across both services, such as statutory holidays) to differentiate the subjects of service disruption.

Similarly, we can expect a delay to always be described using the word delay, a detour by the term detour, and a diversion, using the word diversion. This means that we can filter out unwanted tweets using specific key terms. Nice job, Translink!

> The data used in this chapter is provided within the GitHub solution accompanying this chapter in the `translink_tweet_data.json` file. The scraper script is also provided within the chapter code; in order to leverage it, you'll need to set up a developer account with Twitter. This is easy to achieve; the process is documented here and you can sign up here.

Once we've obtained our tweet data, we know what to do next—we need to clean and regularize the body text! As per *Chapter 6, Text Feature Engineering,* let's run `BeautifulSoup` and `NLTK` over the input data:

```
from bs4 import BeautifulSoup

tweets = BeautifulSoup(train["TranslinkTweets.text"])

tweettext = tweets.get_text()

brown_a = nltk.corpus.brown.tagged_sents(categories= 'a')

tagger = None
for n in range(1,4):
    tagger = NgramTagger(n, brown_a, backoff = tagger)

taggedtweettext = tagger.tag(tweettext)
```

We probably will not need to be as intensive in our cleaning as we were with the troll dataset in the previous chapter. Translink's tweets are highly formulaic and do not include non-ascii characters or emoticons, so the specific "deep cleaning" regex script that we needed to use in *Chapter 6, Text Feature Engineering*, won't be needed here.

This gives us a dataset with lower-case, regularized, and dictionary-checked terms. We are ready to start thinking seriously about what features we ought to build out of this data.

We know that the base method of detecting a service disruption issue within our data is the use of a delay term in a tweet. Delays happen in the following ways:

- At a given location
- At a given time
- For a given reason
- For a given duration

Each of the first three factors is consistently tracked within Translink tweets, but there are some data quality concerns that are worth recognizing.

Location is given in terms of an affected street or station *at 22nd Street*. This isn't a perfect description for our purpose as we're unlikely to be able to turn a street name and route start/end points into a general *affected area* without doing substantial additional work (as no convenient reference exists that allows us to draw a bounding box based on this information).

Time is imperfectly given by the tweet datetime. While we don't have visibility on whether tweets are made within a consistent time from service disruption, it's likely that Translink has targets around service notification. For now, it's sensible to proceed under the assumption that the tweet times are likely to be sufficiently accurate.

The exception is likely to be for long-running issues or problems that change severity (delays that are expected to be minor but which become significant). In these cases, tweets may be delayed until the Translink team recognizes that the issue has become tweet-worthy. The other possible cause of data quality issues is inconsistency in Translink's internal communications; it's possible that engineering or platform teams don't always inform the customer service notifications team at the same speed.

We're going to have to take a certain amount on faith though, as there isn't a huge amount we can do to measure these delay effects without a dataset of real-time, accurate Translink service delays. (If we had that, we'd be using it instead!)

Reasons for Skytrain service delays are consistently described by Translink and can fall into one of the following categories:

- Rail
- Train
- Switch
- Control
- Unknown
- Intrusion
- Medical
- Police
- Power

With each category described within the tweet body using the specific proper term given in the preceding list. Obviously, some of these categories (Police, Power, Medical) are less likely to be relevant as they wouldn't tell us anything useful about road conditions. The rate of train, track, and switch failure may be correlated with detour likelihood; this suggests that we may want to keep those cases for classification purposes.

Meanwhile, bus route service delays contain a similar set of codes, many of which are very relevant to our purposes. These codes are as follows:

- **Motor Vehicle Accident** (**MVA**)
- Construction
- Fire
- Watermain
- Traffic

Encoding these incident types is likely to prove useful! In particular, it's possible that certain service delay types are more impactful than others, increasing the risk of a longer service delay. We'll want to encode service delay types and use them as parameters in our subsequent modeling.

To do this, let's apply a variant of one-hot encoding, which does the following:

- It creates a conditional variable for each of the service risk types and sets all values to zero
- It checks tweet content for each of the service risk type terms
- It sets the relevant conditional variable to 1 for each tweet that contains a specific risk term

This effectively performs one-hot encoding without taking the bothersome intermediary step of creating the factorial variable that we'd normally be processing:

```
from sklearn import preprocessing

enc = preprocessing.OneHotEncoder(categorical_features='all', dtype=
'float', handle_unknown='error', n_values='auto', sparse=True)

tweets.delayencode = enc.transform(tweets.delaytype).toarray()
```

Beyond what we have available to use as a feature on a per-incident basis, we can definitely look at the relationship between service disruption risk and disruption frequency. If we see two disruptions in a week, is a third more likely or less likely?

While these questions are interesting and potentially fruitful, it's usually more prudent to work up a limited feature set and simple model on a first pass than to overengineer a sprawling feature set. As such, we'll run with the initial incidence rate features and see where we end up.

Consumer comments

A major cultural development in 2010 was the widespread use of public online domains for self-expression. One of the happier products of this is the availability of a wide array of self-reported information on any number of subjects, provided we know how to tap into this.

Commute disruptions are frequently occurring events that inspire a personal response, which means that they tend to be quite broadly reported on social media. If we write an appropriate dictionary for key-term search, we can begin using Twitter particularly as a source of timestamped information on traffic and transit issues around the city.

In order to collect this data, we'll make use of a dictionary-based search approach. We're not interested in the majority of tweets from the period in question (and as we're using the RESTful API, there are return limits to consider). Instead, we're interested in identifying tweet data containing key terms related to congestion or delay.

Unfortunately, tweets harvested from a broad range of users tend not to conform to consistent styles that aid analysis. We're going to have to apply some of the techniques we developed in the preceding chapter to break down this data into a more easily analyzed format.

In addition to using a dictionary-based search, we could do some work to narrow the search area down. The most authoritative way to achieve this is to use a bounding box of coordinates as an argument to the Twitter API, such that any related query exclusively returns results gathered from within this area.

As always, on our first pass, we'll keep things simple. In this case, we'll count up the number of traffic disruption tweets in the current period. There is some additional work that we could benefit from doing with this data on subsequent iterations. Just as the Translink data contained clearly-defined delay cause categories, we could try to use specialized dictionaries to isolate delay types based on key terms (for example, a dictionary of construction-related terms and synonyms).

We could also look at defining a more nuanced quantification of disruption tweet rate than a simple count of recent. We could, for instance, look at creating a weighted count feature that increases the impact of multiple simultaneous tweets (potentially indicative of severe disruption) via a nonlinear weighting.

The Bing Traffic API

The next API we're going to tap into is the Bing Traffic API. This API has the advantage of being easily accessed; it's freely available (whereas some competitor APIs sit behind paywalls), returns data, and provides a good level of detail. Among other things, the API returns an incident location code, a general description of the incident, together with congestion information, an incident type code, and start/end timestamps.

Helpfully, the incident type codes provided by this API describe a broad set of incident types, as follows:

1. `Accident.`
2. `Congestion.`
3. `DisabledVehicle.`
4. `MassTransit.`
5. `Miscellaneous.`
6. `OtherNews.`
7. `PlannedEvent.`
8. `RoadHazard.`
9. `Construction.`
10. `Alert.`
11. `Weather.`

Additionally, a severity code is provided with the severity values translated as follows:

1. `LowImpact.`
2. `Minor.`
3. `Moderate.`
4. `Serious.`

One downside, however, is that this API doesn't receive consistent information between regions. Querying in France, for instance, returns codes from multiple other incident types, (I observed 1, 3, 5, 8 for a town in northern France over a period of one month.) but doesn't seem to show every code. In other locations, even less data is available. Sadly, Vancouver tends to show data for codes 9 or 5 exclusively, but even the miscellaneous-coded incidents appear to be construction-related:

```
Closed between Victoria Dr and Commercial Dr - Closed. Construction
work. 5
```

This is a somewhat bothersome limitation. Unfortunately, it's not something that we can easily fix; Bing's API is simply not sourcing all of the data that we want! Unless we pay for a more complete dataset (or an API with fuller data capture is available in your area!), we're going to need to keep working with what we have.

An example of querying this API is as follows:

```
importurllib.request, urllib.error, urllib.parse
import json

latN = str(49.310911)
latS = str(49.201444)
lonW = str(-123.225544)
lonE = str(-122.903931)

url = 'http://dev.virtualearth.net/REST/v1/Traffic/Incidents/'
+latS+','+lonW+','+latN+','+lonE+'?
key='GETYOUROWNKEYPLEASE'

response = urllib.request.urlopen(url).read()
data = json.loads(response.decode('utf8'))
resources = data['resourceSets'][0]['resources']

print('----------------------------------------------------')
print('PRETTIFIED RESULTS')
print('----------------------------------------------------')
for resourceItem in resources:
    description = resourceItem['description']
typeof = resourceItem['type']
    start = resourceItem['start']
    end = resourceItem['end']
print('description:', description);
print('type:', typeof);
print('starttime:', start);
print('endtime:', end);
print('----------------------------------------------------')


This example yields the following data;

----------------------------------------------------
PRETTIFIED RESULTS
----------------------------------------------------
description: Closed between Boundary Rd and PierviewCres - Closed due
to roadwork.
```

```
type: 9
severity 4
starttime: /Date(1458331200000)/
endtime: /Date(1466283600000)/
--------------------------------------------------
description: Closed between Commercial Dr and Victoria Dr - Closed due
to roadwork.
type: 9
severity 4
starttime: /Date(1458327600000)/
endtime: /Date(1483218000000)/
--------------------------------------------------
description: Closed between Victoria Dr and Commercial Dr - Closed.
Construction work.
type: 5
severity 4
starttime: /Date(1461780543000)/
endtime: /Date(1481875140000)/
--------------------------------------------------
description: At Thurlow St - Roadwork.
type: 9
severity 3
starttime: /Date(1461780537000)/
endtime: /Date(1504112400000)/
--------------------------------------------------
```

Even after recognizing the shortcomings of uneven code availability across different geographical areas, the data from this API should provide us with some value. Having a partial picture of traffic disruption incidents still gives us data for a reasonable period of dates. The ability to localize traffic incidents within an area of our own definition and returning data relevant to the current date is likely to help the performance of our model.

Deriving and selecting variables using feature engineering techniques

On our first pass over the input data, we repeatedly made the choice to keep our initial feature set small. Though we saw lots of opportunities in the data, we prioritized viewing an initial result above following up on those opportunities.

It is likely, however, that our first dataset won't help us solve the problem very effectively or hit our targets. In this event, we'll need to iterate over our feature set, both by creating new features and winnowing our feature set to reduce down to the valuable outputs of that feature creation process.

One helpful example involves one-hot encoding and RFE. In this chapter, we'll use one-hot to turn weather data and tweet dictionaries into tensors of m*n size. Having produced m-many new columns of data, we'll want to reduce the liability of our model to be misled by some of these new features (for instance, in cases where multiple features reinforce the same signal or where misleading but commonly-used terms are not cleaned out by the data cleaning processes we described in *Chapter 6, Text Feature Engineering*). This can be done very effectively by RFE, the technique for feature selection that we discussed earlier in this chapter.

In general, it can be helpful to work using a methodology that applies the techniques seen in the last two chapters using an expand-contract process. First, use techniques that can generate potentially valuable new features, such as transformations and encodings, to expand the feature set. Then, use techniques that can identify the most performant subset of those features to remove the features that do not perform well. Throughout this process, test different target feature counts to identify the best available feature set at different numbers of features.

Some data scientists interpret how this is done differently from others. Some will build all of their features using repeated iterations over the feature creation techniques we've discussed, then reduce that feature set—the motivation being that this workflow minimizes the risk of losing data. Others will perform the full process iteratively. How you choose to do this is entirely up to you!

On our initial pass over the input data, then, we have a feature set that looks as follows:

```
{
  'DisruptionInformation': {
    'Date': '15-05-2015',
    'TranslinkTwitter': [{
      'Service': '0',
      'DisruptionIncidentCount': '4'
  }, {
      'Service': '1',
      'DisruptionIncidentCount': '0'
    }]
  },
  'BingTrafficAPI': {
    'NewIncidentCount': '1',
    'SevereIncidentCount': '1',
    'IncidentCount': '3'
  },
  'ConsumerTwitter': {
    'DisruptionTweetCount': '4'
  }
}
```

It's unlikely that this dataset is going to perform well. All the same, let's run it through a basic initial algorithm and get a general idea as to how near our target we are; this way, we can learn quickly with minimal overhead!

In the interest of expedience, let's begin by running a first pass using a very simple regression algorithm. The simpler the technique, the faster we can run it (and often, the more transparent it is to us what went wrong and why). For this reason (and because we're dealing with a regression problem with a continuous output rather than a classification problem), on a first pass we'll work with a simple linear regression model:

```
from sklearn import linear_model

tweets_X_train = tweets_X[:-20]
tweets_X_test = tweets_X[-20:]


tweets_y_train = tweets.target[:-20]
tweets_y_test = tweets.target[-20:]


regr = linear_model.LinearRegression()

regr.fit(tweets_X_train, tweets_y_train)

print('Coefficients: \n', regr.coef_)
print("Residual sum of squares: %.2f" % np.mean((regr.
predict(tweets_X_test) - tweets_y_test) ** 2))

print('Variance score: %.2f' % regr.score(tweets_X_test, tweets_y_
test))

plt.scatter(tweets_X_test, tweets_y_test,  color='black')
plt.plot(tweets_X_test, regr.predict(tweets_X_test),
color='blue',linewidth=3)

plt.xticks(())
plt.yticks(())
plt.show()
```

At this point, our AUC is pretty lousy; we're looking at a model with an AUC of 0.495. We're actually doing worse than our target! Let's print out a confusion matrix to see what this model's doing wrong:

		Prediction	
		TRUE	FALSE
Actual Result	TRUE	1	9
	FALSE	18	136

According to this matrix, it's doing everything not very well. In fact, it's claiming that almost all of the records show no incidents, to the extent of missing 90% of real disruptions!

This actually isn't too bad at all, given the early stage that we're at with our model and our features, as well as the uncertain utility of some of our input data. At the same time, we should expect an incidence rate of 6% (as our training data suggests that incidents have been seen to occur roughly once every 16 commutes). We'd still be doing a little better by guessing that every day will involve a disrupted commute (if we ignore the penalty to our lifestyle entailed by leaving home early each day).

Let's consider what changes we could make in a next pass.

1. First off, we could stand to improve our input data further. We identified a number of new features that we could create from existing sources using a range of transformation techniques.

2. Secondly, we could look at extending our dataset using additional information. In particular, a weather dataset describing both temperature and humidity may help us improve our model.

3. Finally, we could upgrade our algorithm to something with a little more grunt, random forests or SVM being obvious examples. There are good reasons not to do this just yet. The main reason is that we can continue to learn a lot from linear regression; we can compare against earlier results to understand how much value our changes are adding, while retaining a fast iteration loop and simple scoring methods. Once we begin to get minimal returns on our feature preparation, we should consider upgrading our model.

For now, we'll continue to upgrade our dataset. We have a number of options here. We can encode location into both traffic incident data from the Bing API's "description" field and into Translink's tweets. In the case of Translink, this is likely to be more usefully done for bus routes than Skytrain routes (given that we restricted the scope of this analysis to focus solely on traffic commutes).

We can achieve this goal in one of two ways;

- Using a corpus of street names/locations, we can parse the input data and build a one-hot matrix
- We can simply run one-hot encoding over the entire body of tweets and entire set of API data

Interestingly, if we intend to use dimensionality reduction techniques after performing one-hot encoding, we can encode the entire body of both pieces of text information without any significant problems. If features relating to the other words used in tweets and text are not relevant, they'll simply be scrubbed out during RFE.

This is a slightly laissez-faire approach, but there is a subtle advantage. Namely, if there is some other potentially useful content to either data source that we've so far overlooked as a potential feature, this process will yield the added benefit of creating features based on that information.

Let's encode locations in the same way we encoded delay types:

```
from sklearn import preprocessing

enc = preprocessing.OneHotEncoder(categorical_features='all', dtype=
'float', handle_unknown='error', n_values='auto', sparse=True)

tweets.delayencode = enc.transform(tweets.location).toarray()
```

Additionally, we should follow up on our intention to create recent count variables from Translink and Bing maps incident logging. The code for this aggregation is available in the GitHub repository accompanying this chapter!

Rerunning our model with this updated data produced results with a very slight improvement; the predicted variance score rose to 0.56. While not dramatic, this is definitely a step in the right direction.

Next, let's follow up on our second option—adding a new data source that provides weather data.

The weather API

We've previously grabbed data that will help us tell whether commute disruption is happening—reactive data sources that identify existing delays. We're going to change things up a little now, by trying to find data that relates to the causes of delays and congestion. Roadworks and construction information definitely falls into this category (along with some of the other Bing Traffic API codes).

One factor that is often (anecdotally!) tied to increased commute time is bad weather. Sometimes this is pretty obvious; heavy frost or high winds have a clear impact on commute time. In many other cases, though, it's not clear what the strength and nature of the relationship between climatic factors and disruption likelihood is for a given commute.

By extracting pertinent weather data from a source with sufficient granularity and geo coverage, we can hopefully use strong weather signals to help improve our correct prediction of disruption.

For our purposes, we'll use the Yahoo Weather API, which provides a range of temperature, atmospheric, pressure-related, and other climate data, both current and forecasted. We can query the Yahoo Weather API without needing a key or login process, as follows:

```
import urllib2, urllib, json

baseurl = https://query.yahooapis.com/v1/public/yql?

yql_query = "select item.condition from weather.forecast where
woeid=9807"
yql_url = baseurl + urllib.urlencode({'q':yql_query}) + "&format=json"
result = urllib2.urlopen(yql_url).read()
data = json.loads(result)
print data['query']['results']
```

To get an understanding for what the API can provide, replace `item.condition` (in what is fundamentally an embedded SQL query) with `*`. This query outputs a lot of information, but digging through it reveals valuable information, including the current conditions:

```
{
   'channel': {
     'item': {
       'condition': {
          'date': 'Thu, 14 May 2015 03:00 AM PDT', 'text': 'Cloudy',
'code': '26', 'temp': '46'
       }
     }
   }
}
```

7-day forecasts containing the following information:

```
{
    'item': {
      'forecast': {
         'code': '39', 'text': 'Scattered Showers', 'high': '60',
'low': '44', 'date': '16 May 2015', 'day': 'Sat'
      }
   }
}
```

And other current weather information:

```
'astronomy': {
      'sunset': '8:30 pm', 'sunrise': '5:36 am'

   'wind': {
      'direction':  '270', 'speed': '4', 'chill': '46'
```

For the purpose of building a training dataset, we extracted data on a daily basis via an automated script that ran from May 2015 to January 2016. The forecasts may not be terribly useful to us as it's likely that our model will rerun over current data on a daily basis rather than being dependent on forecasts. However, we will definitely make use of the `wind.direction`, `wind.speed`, and `wind.chill` variables, as well as the `condition.temperature` and `condition.text` variables.

In terms of how to further process this information, one option jumps to mind. One-hot encoding of weather tags would enable us to use weather condition information as categorical variables, just as we did in the preceding chapter. This seems like a necessary step to take. This significantly inflates our feature set, leaving us with the following data:

```
{
  'DisruptionInformation': {
    'Date': '15-05-2015',
    'TranslinkTwitter': [{
      'Service': '0',
      'DisruptionIncidentCount': '4'
    }, {
      'Service': '1',
      'DisruptionIncidentCount': '0'
    }]
  },
  'BingTrafficAPI': {
    'NewIncidentCount': '1',
    'SevereIncidentCount': '1',
```

```
      'IncidentCount': '3'
    },
    'ConsumerTwitter': {
      'DisruptionTweetCount': '4'
    },
    'YahooWeather':{
      'temp: '45'
      'tornado': '0',
      'tropical storm': '0',
      'hurricane': '0',
      'severe thunderstorms': '0',
      'thunderstorms': '0',
      'mixed rain and snow': '0',
      'mixed rain and sleet': '0',
      'mixed snow and sleet': '0',
'freezing drizzle': '0',
'drizzle': '0',
'freezing rain': '0',
'showers': '0',
'snow flurries': '0',
'light snow showers': '0',
'blowing snow': '0',
'snow': '0',
'hail': '0',
'sleet': '0',
'dust': '0',
'foggy': '0',
'haze': '0',
'smoky': '0',
'blustery': '0',
'windy': '0',
'cold': '0',
'cloudy': '1',
'mostly cloudy (night)': '0',
'mostly cloudy (day)': '0',
'partly cloudy (night)': '0',
'partly cloudy (day)': '0',
'clear (night)': '0',
'sunny': '0',
'fair (night)': '0',
'fair (day)': '0',
'mixed rain and hail': '0',
'hot': '0',
'isolated thunderstorms': '0',
```

```
'scattered thunderstorms': '0',
'scattered showers': '0',
'heavy snow': '0',
'scattered snow showers': '0',
'partly cloudy': '0',
'thundershowers': '0',
'snow showers': '0',
'isolated thundershowers': '0',
'not available': '0',
}
```

It's very likely that a lot of time could be valuably sunk into further enriching the weather data provided by the Yahoo Weather API. For the first pass, as always, we'll remain focused on building a model that takes the features that we described previously.

> It's definitely worth considering how we would do further work with this data. In this case, it's important to distinguish between cross-column data transformations and cross-row transformations.
>
> A cross-column transformation is one where variables from different features in the same input case were transformed based on one another. For instance, we might take the start date and end date of a case and use it to calculate the duration. Interestingly, the majority of the techniques that we've studied in this book won't gain a lot from many such transformations. Most machine learning techniques capable of drawing nonlinear decision boundaries tend to encode relationships between variables in their modeling of a dataset. Deep learning techniques often take this capability a step further. This is part of the reason that some feature engineering techniques (particularly basic transformations) add less value for deep learning applications.
>
> Meanwhile, a cross-row transformation is typically an aggregation. The central tendency of the last n-many duration values, for instance, is a feature that can be derived by an operation over multiple rows. Naturally, some features can be derived by a combination of column-wise and row-wise operations. The interesting thing about cross-row transformations is that it's usually quite unlikely that a model will train to recognize them, meaning that they tend to continue to add value in very particular contexts.
>
> The reason that this information is relevant, of course, is that recent weather is a context in which features derived from cross-row operations might add new information to our model. Change in barometric pressure or temperature over the last n hours, for instance, might be a more useful variable than the current pressure or temperature. (Particularly, when that our model is intended to predict commutes to take place later in the same day!)
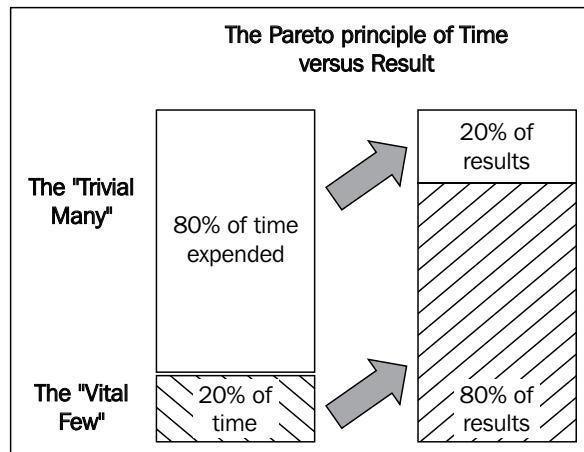
The next step is to rerun our model. This time, our AUC is a little higher; we're scoring *0.534*. Looking at our confusion matrix, we're also seeing improvements:

Actual Result		Prediction	
		TRUE	FALSE
	TRUE	3	7
	FALSE	22	132

If the issues are linked to weather factors, continuing to pull weather data is a good idea; setting this solution up to run over an extended period will gradually gather longitudinal inputs from each source, gradually giving us much more reliable predictions.

At this point, we're only a short distance away from our MVP target. We can continue to extend our input dataset, but the smart solution is to find another way to approach the problem. There are two actions that we can meaningfully take.

Being human, data scientists tend to think in terms of simplifying assumptions. One of these that crops up quite frequently is basically an application of the Pareto principle to cost/benefit analysis decisions. Fundamentally, the Pareto principle states that for many events, roughly 80% of the value or effect comes from roughly 20% of the input effort, or cause, obeying what's referred to as a Pareto distribution. This concept is very popular in software engineering contexts among others, as it can guide efficiency improvements.

The Pareto principle of Time versus Result

The "Trivial Many"

80% of time expended

20% of results

The "Vital Few"

20% of time

80% of results

> To apply this theory to the current case, we know that we could spend more time finessing our feature engineering. There are techniques that we haven't applied and other features that we could create. However, at the same time, we know that there are entire areas that we haven't touched: external data searches and model changes, particularly, which we could quickly try. It makes sense to explore these cheap but potentially impactful options on our next pass before digging into additional dataset preparation.

During our exploratory analysis, we noticed that some of our variables are quite sparse. It wasn't immediately clear how helpful they all were (particularly for stations where fewer incidents of a given type occurred).

Let's test out our variable set using some of the techniques that we worked with earlier in the chapter. In particular, let's apply `Lasso` to the problem of reducing our feature set to a performant subset:

```
fromsklearn.preprocessing import StandardScaler


scaler = StandardScaler()
X = scaler.fit_transform(DisruptionInformation["data"])
Y = DisruptionInformation["target"]
names = DisruptionInformation["feature_names"]

lasso = Lasso(alpha=.3)
lasso.fit(X, Y)

print "Lasso model: ", pretty_print_linear(lasso.coef_, names, sort =
True)
```

This output is immediately valuable. It's obvious that many of the weather features (either through not showing up sufficiently often or not telling us anything useful when they do) are adding nothing to our model and should be removed. In addition, we're not getting a lot of value from our traffic aggregates. While these can remain in for the moment (in the hope that gathering more data will improve their usefulness), for our next pass we'll rerun our model without the poorly-scoring features that our use of LASSO has revealed.

There is one fairly cheap additional change, which we ought to make: we should upgrade our model to one that can fit nonlinearly and thus can fit to approximate any function. This is worth doing because, as we observed, some of our features showed a range of skewed distributions indicative of a nonlinear underlying trend. Let's apply a random forest to this dataset:

```
fromsklearn.ensemble import RandomForestClassifier,
ExtraTreesClassifier
rf = RandomForestRegressor(n_jobs = 3, verbose = 3, n_estimators=20)
rf.fit(DisruptionInformation_train.targets,DisruptionInformation_
train.data)

r2 = r2_score(DisruptionInformation.data, rf.predict(DisruptionInforma
tion.targets))
mse = np.mean((DisruptionInformation.data - rf.predict(DisruptionInfor
mation.targets))**2)

pl.scatter(DisruptionInformation.data, rf.predict(DisruptionInformati
on.targets))
pl.plot(np.arange(8, 15), np.arange(8, 15), label="r^2=" + str(r2),
c="r")
pl.legend(loc="lower right")
pl.title("RandomForest Regression with scikit-learn")
pl.show()
```

Let's return again to our confusion matrix:

		Prediction	
		TRUE	FALSE
Actual Result	TRUE	4	6
	FALSE	15	134

At this point, we're doing fairly well. A simple upgrade to our model has yielded significant improvements, with our model correctly identifying almost 40% of commute delay incidents (enough to start to be useful to us!), while misclassifying a small amount of cases.

Frustratingly, this model would still be getting us out of bed early incorrectly more times than it would correctly. The gold standard, of course, would be if it were predicting more commute delays than it was causing false (early) starts! We could reasonably hope to achieve this target if we continue to gather feature data over a sustained period; the main weakness of this model is that it has very few cases to sample from, given the rarity of commute disruption events.

We have, however, succeeded in gathering and marshaling a range of data from different sources in order to create a model from freely-available data that yields a recognizable, real-world benefit (reducing the amount of late arrivals at work by 40%). This is definitely an achievement to be happy with!

Further reading

My suggested go-to introduction to feature selection is Ando Sabaas' four-part exploration of a broad range of feature selection techniques. It's full of Python code snippets and informed commentary. Get started at `http://blog.datadive.net/ selecting-good-features-part-i-univariate-selection/`.

For a discussion on feature selection and engineering that ranges across materials in chapters 6 and 7, consider Alexandre Bourhard-Côté's slides at `http://people.eecs.berkeley.edu/~jordan/courses/294-fall09/lectures/ feature/slides.pdf`. Also consider reviewing Jeff Howbert's slides at `http:// courses.washington.edu/css490/2012.Winter/lecture_slides/05a_feature_ creation_selection.pdf`.

There is a shortage of thorough discussion of feature creation, with a lot of available material discussing either dimensionality reduction techniques or very specific feature creation as required in specific domains. One way to get a more general understanding of the range of possible transformations is to read code documentation. A decent place to build on your existing knowledge is Spark ML's feature-transformation algorithm documentation at `https://spark.apache.org/ docs/1.5.1/ml-features.html#feature-transformers`, which describes a broad set of possible transformations on numerical and text features. Remember, though, that feature creation is often problem-specific, domain-specific, and a highly creative process. Once you've learned a range of technical options, the trick is in figuring out how to apply these techniques to the problem at hand!

For readers with an interest in hyperparameter optimization, I recommend that you read Alice Zheng's posts on Turi's blog as a great place to start: `http://blog.turi. com/how-to-evaluate-machine-learning-models-part-4-hyperparameter- tuning`.

I also find the scikit-learn documentation to be a useful reference for grid search specifically: `http://scikit-learn.org/stable/modules/grid_search.html`.

Summary

In this chapter, you learned and applied a set of techniques that enable us to effectively build and finesse datasets for machine learning, starting from very little initial data. These powerful techniques enable a data scientist to turn seemingly shallow datasets into opportunities. We demonstrated this power using a set of customer service tweets to create a travel disruption predictor.

In order to take that solution into production, though, we'd need to add some functionality. Removing some locations in the penultimate step was a questionable decision; if this solution is intended to identify journey disruption risk, then removing locations seems like a non-starter! This is particularly true given that we do not have year-round data and so cannot identify the effect of seasonal or longitudinal trends (like extended maintenance works or a scheduled station closure). We were a little hasty in removing these elements and a better solution would be to retain them for a longer period.

Following on from these concerns, we should recognize the need to start building some dynamism into our solution. When spring rolls around and our dataset starts to contain new climate conditions, it is entirely likely that our model will fail to adapt as effectively. In the next chapter, we will be looking at building more sophisticated model ensembles and discuss methods of building robustness into your model solutions.

8
Ensemble Methods

As we progressed through the earlier chapters of this book, you learned how to apply a number of new techniques. We developed our use of several advanced machine learning algorithms and acquired a broad range of companion techniques used to enhance your use of learning techniques via more effective feature selection and preparation. This chapter seeks to enhance your existing technique set using ensemble methods: techniques that bind multiple different models together to solve a real-world problem.

Ensemble techniques have become a fundamental part of the data scientist's toolset. The use of ensembles has become common practice in competitive machine learning contexts, and ensembles are now considered an indispensable tool in many contexts. The techniques that we'll develop in this chapter give our models an edge in performance, while increasing their robustness to underlying data change.

We'll examine a series of ensembling options, discussing both the code and application of these techniques. We'll color this explanation with guidance and reference to real-world applications, including the models created by successful Kagglers.

The development of any of the models that we reviewed in this title allows us to solve a wide range of data problems, but applying our models to production contexts raises an additional set of problems. Our solutions are still vulnerable to changes in the underlying observations. Whether this is expressed in a different population of individuals, in temporal variations (for example, seasonal changes in the phenomenon being captured) or by other changes to the underlying conditions, the end result is often the same—the models that worked well in the conditions they were trained against are frequently unable to generalize and continue to perform well as time passes.

The final section of this chapter describes methodologies to transfer the techniques from this book to operational environments and the kinds of additional monitoring and support you should consider if your intended applications have to be resilient to change.

Introducing ensembles

"This is how you win ML competitions: you take other peoples' work and ensemble them together."

– Vitaly Kuznetsov NIPS2014

In the context of machine learning, an ensemble is a set of models that is used to solve a shared problem. An ensemble is made up of two components: a set of models and a set of decision rules that govern how the results of those models are combined into a single output.

Ensembles offer a data scientist the ability to construct multiple solutions for a given problem and then combine these into a single final result that draws from the best elements of each input solution. This provides robustness against noise, which is reflected in more effective training against an initial dataset (leading to lower levels of overfitting and reductions in training error) and against data change of the kinds discussed in the preceding section.

It is no exaggeration to say that ensembles are the most important recent development in machine learning.

In addition, ensembles enable greater flexibility in how one solves for a given problem, in that they enable the data scientist to test different parts of a solution and resolve issues specific to subsets of the input data or parts of the models in use, without completely retuning the whole model. As we'll see, this can make life easier!

Ensembles are typically considered as falling into one of several classes, based on the nature of the decision rules used. The key ensemble types are as follows:

- **Averaging methods**: They develop models in parallel and then use averaging or voting techniques to develop a combined estimator
- **Stacking (or Blending) methods**: They use the weighted output of multiple classifiers as inputs to a next-layer model
- **Boosting methods**: They involve building models in sequence where each added model aims to improve the score of the combined estimator

Given the importance and utility of both of these classes of the ensemble method, we'll treat each one in turn: discussing theory, algorithm options, and real-world examples.

Understanding averaging ensembles

Averaging ensembles have a long and rich history in the physical sciences and statistical modeling, seeing a common application in many contexts including molecular dynamics and audio signal processing. Such ensembles are typically seen as almost exactly replicated cases of a given system. The average (mean) values of and variance between cases in this system are key values for the system as a whole.

In a machine learning context, an averaging ensemble is a collection of models that train on the same dataset, whose results are aggregated in a range of ways. Depending on implementation goals, an averaging ensemble can bring several benefits.

Averaging ensembles can be used to reduce the variability of a model's performance. One common method involves creating multiple model configurations that take different parameter subsets as input. Techniques that take this approach are referred to collectively as bagging algorithms.

Using bagging algorithms

Different bagging implementations will operate differently but share the common property of taking random subsets of the feature space. There are four main types of the bagging approach. Pasting draws random subsets of the samples without replacement. When this is done with replacement, then the approach is simply called **bagging**. Pasting is typically computationally cheaper than bagging and can yield similar results in simpler applications.

When samples are taken feature-wise, the method is known as **random subspaces**. Random subspace methods provide a slightly different capability; they essentially reduce the need for extensive, highly optimized feature selection. Where such activities typically lead to a single model with optimized input, random subspaces allow the use of multiple configurations in parallel, with a flattening of the variance of any one solution.

> While the use of an ensemble to reduce the variability in model performance may sound like a performance hit (the natural response might be but why not just pick the single best performing model in the ensemble?), there are big advantages to this approach.
>
> Firstly, as discussed, averaging improves the ability of your model set to adapt to unfamiliar noise (that is, it reduces overfitting). Secondly, an ensemble can be used to target different elements of the input dataset to model effectively. This is a common approach in competitive machine learning contexts, where a data scientist will iteratively adjust the ensemble based on the results of classification and particular types of failure cases. In some cases, this is an exhaustive process involving the inspection of model results (commonly as part of a normal, iterative model development process) but many data scientists prefer techniques or a solution that they will implement first.

Random subspaces can be a very powerful approach, particularly if it's possible to use multiple subspace sizes and exhaustively check feature combinations. The cost of random subspace methods increases nonlinearly with the size of your dataset and, beyond a certain point, it will become costly to test every configuration of parameters for multiple subspace sizes.

Finally, an ensemble's estimators may be created from subsets drawn from both samples and features, in a method known as **random patches**. On a like-for-like case, the performance of random patches is usually around the same level as that of random subspace techniques with significantly reduced memory consumption.

As we've discussed the theory behind bagging ensembles, let's look at how we go about implementing one. The following code describes a random patches classifier implemented using sklearn's `BaggingClassifier` class:

```
from sklearn.cross_validation import cross_val_score
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)
X = data
y = digits.target


bagging = BaggingClassifier(KNeighborsClassifier(), max_samples=0.5,
max_features=0.5)
scores = cross_val_score(bagging, X, y)
```

```
mean = scores.mean()
print(scores)
print(mean)
```

As with many sklearn classifiers, the core code needed is very straightforward; the classifier is initialized and used to score the dataset. Cross-validation (via `cross_val_score`) adds no meaningful complexity.

This bagging classifier used a **K-Nearest Neighbors** (**KNN**) classifier (`KNeighboursClassifier`) as a base, with feature-wise and case-wise sampling rates each set to 50%. This outputs very strong results against the digits dataset, correctly classifying a mean of 93% of cases after cross-validation:

```
[ 0.94019934  0.92320534  0.9295302 ]


0.930978293043
```

Using random forests

An alternative set of averaging ensemble techniques is referred to collectively as random forests. Perhaps the most successful ensemble technique used by competitive data scientists, random forests develop parallel sets of decision tree classifiers. By introducing two main sources of randomness to the classifier construction, the forest ends up containing diverse trees. The data that is used to build each tree is sampled with replacement from the training set, while the tree creation process no longer uses the best split from all features, instead choosing the best split from a random subset of the features.

Random forests can be easily called using the `RandomForestClassifier` class in `sklearn`. For a simple example, consider the following:

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

clf = RandomForestClassifier(n_estimators=10)
```

```
clf = clf.fit(data, labels)
scores = clf.score(data,labels)
print(scores)
```

The scores output by this ensemble, 0.999, are difficult to beat. Indeed, we haven't seen performance at this level from any of the individual models we employed in preceding chapters.

A variant of random forests, called **extremely randomized trees (ExtraTrees)**, uses the same random subset of features method in selecting the best split at each branch in the tree. However, it also randomizes the discrimination threshold; where a decision tree normally chooses the most effective split between classes, ExtraTrees split at a random value.

Due to the relatively efficient training of decision trees, a random forest algorithm can potentially support a large number of varied trees with the effectiveness of the classifier improving as the number of nodes increases. The randomness introduced provides a degree of robustness to noise or data change; like the bagging algorithms we reviewed earlier, however, this gain typically comes at the cost of a slight drop in performance. In the case of ExtraTrees, the robustness may increase further while the performance measure improves (typically a bias value reduces).

The following code describes how ExtraTrees work in practice. As with our random subspace implementation, the code is very straightforward. In this case, we'll develop a set of models to compare how ExtraTrees shape up against tree and random forest approaches:

```
from sklearn.cross_validation import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)
X = data
y = digits.target

clf = DecisionTreeClassifier(max_depth=None, min_samples_split=1,
    random_state=0)
scores = cross_val_score(clf, X, y)
print(scores)

clf = RandomForestClassifier(n_estimators=10, max_depth=None,
```

```
    min_samples_split=1, random_state=0)
scores = cross_val_score(clf, X, y)
print(scores)

clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
    min_samples_split=1, random_state=0)
scores = cross_val_score(clf, X, y)
print(scores)
```
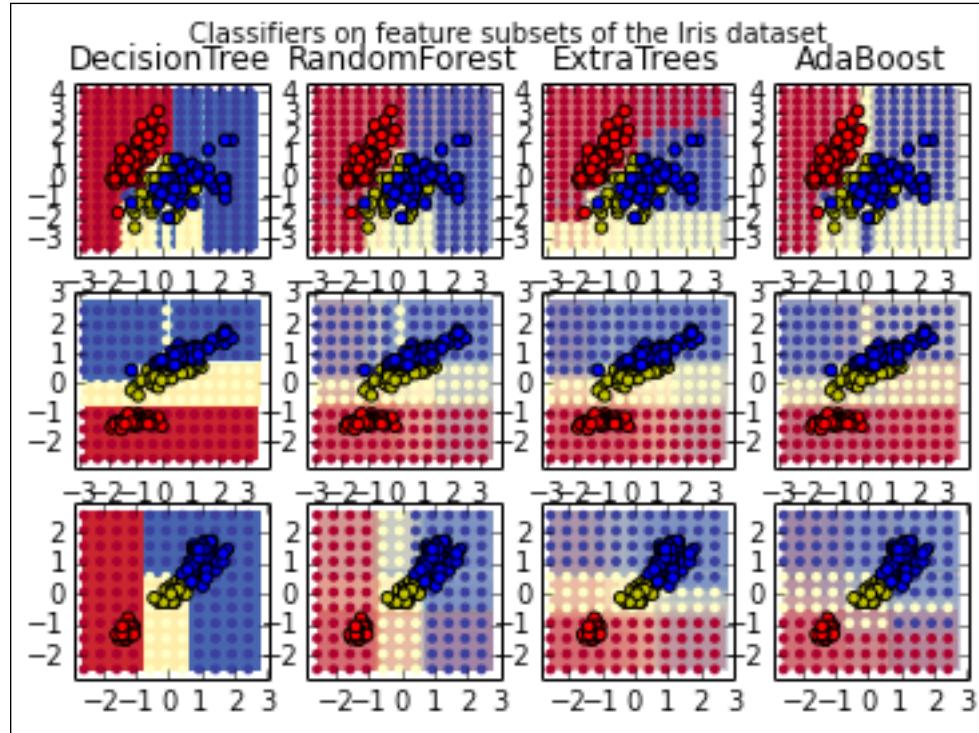
The scores, respectively, are as follows:

```
[ 0.74252492   0.82136895   0.75671141]
[ 0.88372093   0.9015025    0.8909396 ]
[ 0.91694352   0.93489149   0.91778523]
```

Given that we're working with entirely tree-based methods here, the score is simply the proportion of correctly-labeled cases. We can see here that there isn't much in it between the two forest methods, which both perform strongly with mean scores of *0.9*. In this example, random forest actually wins out marginally (on the order of an *0.002* increase) over ExtraTrees, while both techniques substantially outperform the basic decision tree, whose mean score sits at *0.77*.

One drawback when working with random forests (especially as the size of the forest increases) is that it can be hard to review the effectiveness of, or tune, a given implementation. While individual trees are extremely easy to work with, the sheer number of trees in a developed ensemble and the obfuscation created by random splitting can make it challenging to refine a random forest implementation. One option is to begin looking at the decision boundaries that individual models draw. By contrasting the models within one's ensemble, it becomes easier to identify where one model performs better at dividing classes than others.

In this example, for instance, we can easily see how our models perform at a high level without digging into specific details:

While it can be challenging to understand beyond a simple level (using high-level plots and summary scores) how a random forest implementation is performing, the hardship is worthwhile. Random forests perform very strongly with only a minimal cost in additional computation. They are very often a good technique to throw at a problem during the early stages, while one is still determining an angle of attack, because their ability to yield strong results fast can provide a useful benchmark. Once you know how a random forest implementation performs, you can begin to optimize and extend your ensemble.

To this end, we should continue exploring the different ensemble techniques so as to further build out our toolkit of ensembling options.

Applying boosting methods

Another approach to ensemble creation is to build boosting models. These models are characterized by their use of multiple models in sequence to iteratively "boost" or improve the performance of the ensemble.

Boosting models frequently use a series of weak learners, models that provide only marginal gain compared to random guessing. At each iteration, a new weak learner is trained on an adjusted dataset. Over multiple iterations, the ensemble is extended with one new tree (whichever tree optimized the ensemble performance score) at each iteration.

Perhaps the most well-known boosting method is **AdaBoost**, which adjusts the dataset at each iteration by performing the following actions:

- Selecting a decision stump (a shallow, often one-level decision tree, effectively the most significant decision boundary for the dataset in question)
- Increasing the weighting of cases that the decision stump labeled incorrectly, while reducing the weighting of correctly labeled cases

This iterative weight adjustment causes each new classifier in the ensemble to prioritize training the incorrectly labeled cases; the model adjusts by targeting highly-weighted data points. Eventually, the stumps are combined to form a final classifier.

AdaBoost can be used both in classification and regression contexts and achieves impressive results. The following example shows an AdaBoost implementation in action on the `heart` dataset:

```python
import numpy as np

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets.mldata import fetch_mldata
from sklearn.cross_validation import cross_val_score

n_estimators = 400
# A learning rate of 1. may not be optimal for both SAMME and SAMME.R
learning_rate = 1.

heart = fetch_mldata("heart")
X = heart.data
y = np.copy(heart.target)
```

```
y[y==-1]=0

X_test, y_test = X[189:], y[189:]
X_train, y_train = X[:189], y[:189]

dt_stump = DecisionTreeClassifier(max_depth=1, min_samples_leaf=1)
dt_stump.fit(X_train, y_train)
dt_stump_err = 1.0 - dt_stump.score(X_test, y_test)

dt = DecisionTreeClassifier(max_depth=9, min_samples_leaf=1)
dt.fit(X_train, y_train)
dt_err = 1.0 - dt.score(X_test, y_test)

ada_discrete = AdaBoostClassifier(
    base_estimator=dt_stump,
    learning_rate=learning_rate,
    n_estimators=n_estimators,
    algorithm="SAMME")
ada_discrete.fit(X_train, y_train)

scores = cross_val_score(ada_discrete, X_test, y_test)
print(scores)
means = scores.mean()
print(means)
```
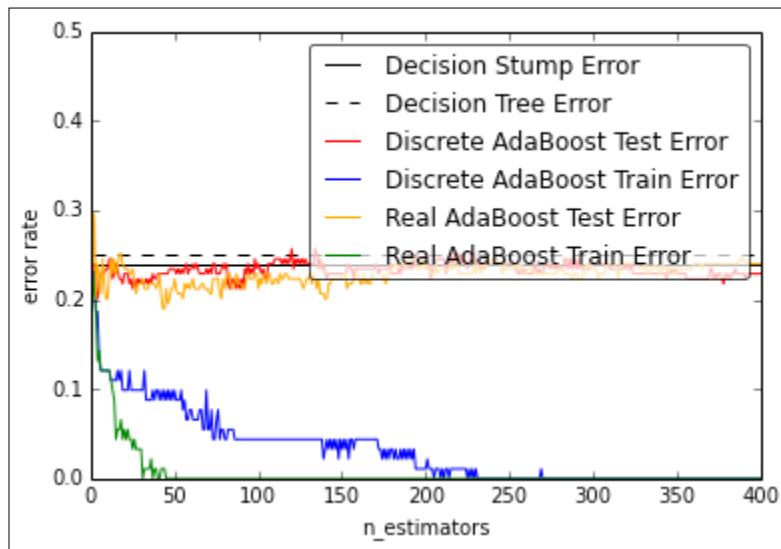
In this case, the `n_estimators` parameter dictates the number of weak learners used; in the case of averaging methods, adding estimators will always reduce the bias of your model, but will increase the probability that your model has overfit its training data. The `base_estimator` parameter can be used to define different weak learners; the default is decision trees (as training a weak tree is straightforward, one can use stumps, very shallow trees). When applied to the `heart` dataset, as in this example, AdaBoost achieved correct labeling in just over 79% of cases, a reasonably solid performance for a first pass:

```
[ 0.77777778  0.81481481  0.77777778]
```

```
0.79012345679
```

Boosting models provide a significant advantage over averaging models; they make it much easier to create an ensemble that identifies problem cases or types of problem cases and address them. A boosting model will usually target the easiest to predict cases first, with each added model fitting against a subset of the remaining incorrectly predicted cases.

One resulting risk is that a boosting model begins to overfit (in the most extreme case, you can imagine ensemble components that have fit to specific cases!) the training data. Managing the correct amount of ensemble components is a tricky problem but thankfully we can resort to a familiar technique to resolve it. In *Chapter 1*, *Unsupervised Machine Learning*, we discussed a visual heuristic called the **elbow method**. In that case, the plot was of *K* (the number of means), against a performance measure for the clustering implementation. In this case, we can employ an analogous process using the number of estimators (*n*) and the bias or error rate for the ensemble (which we'll call e). For a range of different boosting estimators, we can plot their outputs as follows:

By identifying a point at which the curve has begun to level off, we can reduce the risk that our model has overfit, which becomes increasingly likely as the curve begins to level off. This is true for the simple reason that as the curve levels, it necessarily means that the added gains from each new estimator are the correct classification of fewer and fewer cases!

Part of the appeal of a visual aid of this kind is that it enables us to get a feel for how likely our solution is to be overfitting. We can (and should!) be applying validation techniques wherever we can, but in some cases (for example, when aiming to hit a particular MVP target for a model implementation, whether that be informed by use cases or the distribution of scores on the Kaggle public leaderboard), we may be tempted to press forward with a performant implementation. Understanding exactly how attenuated the gains we're receiving are as we add each new estimator is critical to understanding the risk of overfitting.

Using XGBoost

In mid-2015, a new algorithm to solve structured machine learning problems, XGboost, has taken the competitive data science world by storm. **Extreme Gradient Boosting** (**XGBoost**) is a well-written, performant library that provides a generalized boosting algorithm (Gradient Boosting).

XGBoost works much like AdaBoost with one key difference—the means by which the model is improved is different.

At each iteration, XGBoost is seeking to improve the performance of the existing model set by reducing the residuals (the differences between targets and label predictions) of that ensemble. Every iteration, the model added is selected based on whether it is most able to reduce the existing ensemble's residuals. This is analogous to gradient descent (where a function is iteratively minimized by moving against a loss gradient); hence, the name Gradient Boosting.

Gradient Boosting has proven to be highly successful in recent Kaggle contests, where it has supported the winners of the CrowdFlower Competition and Microsoft Malware Classification Challenge, along with many other structured data competitions in the final half of 2015.

To apply XGBoost, let's grab the XGBoost library. The best way to get this is via `pip`, with the `pip install xgboost` command on the command line. For Windows users, `pip` installation is currently (late 2015) disabled on Windows. For your benefit, a cold copy of XGBoost is available in the `Chapter 8` folder of this book's GitHub repository.

Applying XGBoost is fairly straightforward. In this case, we'll apply the library to a multiclass classification task, using the UCI Dermatology dataset. This dataset contains an age variable and a large number of categorical variables. An example row of data looks like this:

```
3,2,0,2,0,0,0,0,0,0,0,1,2,0,2,1,1,1,0,0,0,1,0,0,0,0,0,0,0,0,1,0,10,2
```

A small number of age values (penultimate feature) are missing, encoded by `?`. The objective in working with this dataset is to correctly classify one of six different skin conditions, per the following class distribution:

```
       Database:  Dermatology


       Class code:   Class:                   Number of instances:
       1             psoriasis      112
       2             seboreic dermatitis           61
       3             lichen planus                 72
```

4	pityriasis rosea	49
5	cronic dermatitis	52
6	pityriasis rubra pilaris	20

We'll begin applying XGBoost to this problem by loading up the data and dividing it into test and train cases via a 70/30 split:

```
import numpy as np
import xgboost as xgb

data = np.loadtxt('./dermatology.data', delimiter=',',converters={33:
lambda x:int(x == '?'), 34: lambda x:int(x)-1 } )
sz = data.shape

train = data[:int(sz[0] * 0.7), :]
test = data[int(sz[0] * 0.7):, :]


train_X = train[:,0:33]
train_Y = train[:, 34]


test_X = test[:,0:33]
test_Y = test[:, 34]
```

At this point, we initialize and parameterize our model. The `eta` parameter defines the step size shrinkage. In gradient descent algorithms, it's very common to use a shrinkage parameter to reduce the size of an update. Gradient descent algorithms have a tendency (especially close to convergence) to zigzag back and forth over the optimum; using a shrinkage parameter to downscale the size of a change makes the effect of gradient descent more precise. A common (and default) scaling value is `0.3`. In this example, `eta` has been set to `0.1` for even greater precision (at the possible cost of more iterations).

The `max_depth` parameter is intuitive; it defines the maximum depth of any tree in the example. Given six output classes, six is a reasonable value to begin with. The `num_round` parameter defines how many rounds of Gradient Boosting the algorithm will perform. Again, you typically require more rounds for a multiclass problem with more classes. The `nthread` parameter, meanwhile, defines how many CPU threads the code will run over.

The `DMatrix` structure used here is purely for the training speed and memory optimization. It's generally a good idea to use these while using XGBoost; they can be built from `numpy.arrays`. Using `DMatrix` enables the `watchlist` functionality, which unlocks some advanced features. In particular, `watchlist` allows us to monitor the evaluation results on all the data in the list provided:

```
xg_train = xgb.DMatrix( train_X, label=train_Y)
xg_test = xgb.DMatrix(test_X, label=test_Y)

param = {}

param['objective'] = 'multi:softmax'

param['eta'] = 0.1
param['max_depth'] = 6
param['nthread'] = 4
param['num_class'] = 6

watchlist = [ (xg_train,'train'), (xg_test, 'test') ]
num_round = 5
bst = xgb.train(param, xg_train, num_round, watchlist );
```

We train our model, `bst`, to generate an initial prediction. We then repeat the training process to generate a prediction with `softmax` enabled (via `multi:softprob`):

```
pred = bst.predict( xg_test );

print ('predicting, classification error=%f' % (sum( int(pred[i]) !=
test_Y[i] for i in range(len(test_Y))) / float(len(test_Y)) ))


param['objective'] = 'multi:softprob'
bst = xgb.train(param, xg_train, num_round, watchlist );

yprob = bst.predict( xg_test ).reshape( test_Y.shape[0], 6 )
ylabel = np.argmax(yprob, axis=1)

print ('predicting, classification error=%f' % (sum( int(ylabel[i]) !=
test_Y[i] for i in range(len(test_Y))) / float(len(test_Y)) ))
```

Using stacking ensembles

The traditional ensembles that we saw earlier in this chapter all shared a common design philosophy: they involve multiple classifiers trained to fit a set of target labels and involve the models themselves being applied to generate some meta-function through strategies including model voting and boosting.
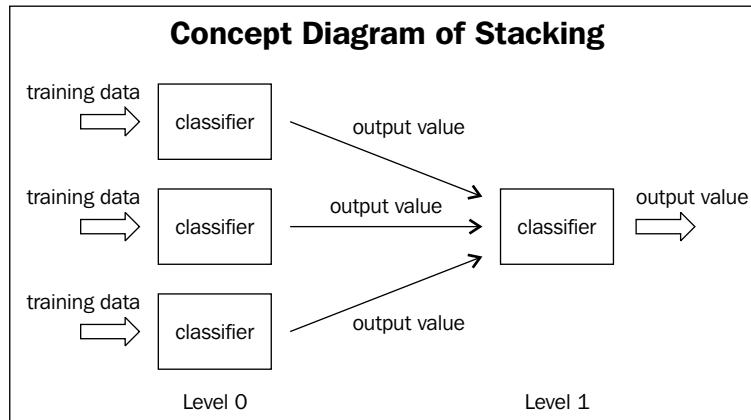
There is an alternative design philosophy as regards ensemble creation, known as stacking or, alternatively, as blending. Stacking involves multiple layers of models in a configuration where the output of one layer of models is used as training data for a model at the next layer. It's possible to blend hundreds of different models successfully.

Stacking ensembles can also make up the blended set of features at a layer's output from multiple sub-blends (sometimes called **blend-of-blends**). To add to the fun, it's also possible to also extract particularly effective parameters from the models of a stacking ensemble and use them as meta-features, within blends or sub-blends at different levels.

All of this combines to make stacking ensembles a very powerful and extensible technique. The winners of the Kaggle Netflix prize (and associated $1 million award) used stacking ensembles over hundreds of features to great effect. They used several additional tricks to improve the effectiveness of their prediction:

- They trained and optimized their ensemble while holding out some data. They then retrained using the held-out data and again optimized before applying their model to the test dataset. This isn't an uncommon practice, but it yields good results and is worth keeping in mind.

- They trained using gradient descent and RMSE as the performance function. Crucially, they used the RMSE of the ensemble, rather than that of any of the models, as the relevant performance indicator (the measure of residuals). This should be considered a healthy practice whenever working with ensembles.

- They used model combinations that are known to improve on the residuals of other models. Neighborhood-based approaches, for instance, improve on the residuals of the RBM, which we examined earlier in this book. By getting to know the relative strengths and weaknesses of your machine learning algorithms, you can find ideal ensemble configurations.

- They calculated the residuals of their blend using k-fold cross-validation, another technique that we explored and applied earlier in this book. This helped overcome the fact that they'd trained their blend's constituent models using the same dataset as the resulting blend.

The main point to take away from the highly customized nature of the **Pragmatic Chaos** model used to win the Netflix prize is that a first-class model is usually the product of intensive iteration and some creative network configuration changes. The other key takeaway is that the basic architectural pattern of a stacking ensemble is as follows:

Now that you've learned the fundamentals of how the stacking ensemble work, let's try applying them to solve data problems. To get us started, we'll use the `blend.py` code provided in the GitHub repository accompanying `Chapter 8,`. Versions of this blending code have been used by highly-scoring Kagglers across multiple contests.

To begin with, we'll examine how stacking ensembles can be applied to attack a real data science problem: the Kaggle contest *Predicting a Biological Response* aimed to build as effective a model as possible in order to predict the biological response of molecules given their chemical properties. We'll be looking at one particularly successful entry in this competition to understand how stacking ensembles can work in practice.

In this dataset, each row represents a molecule, while each of the 1,776 features describe characteristics of the molecule in question. The goal was to predict a binary response from the molecule in question, given these properties.

The code that we'll be applying comes from a competitor in that tournament who used a stacking ensemble to combine five classifiers: two differently configured random forest classifiers, two extra trees classifiers, and a gradient boosting classifier, which helps to yield slightly differentiated predictions from the other four components.

The duplicated classifiers were provided with different split criteria. One used the **Gini Impurity** (gini), a measure of how often a random record would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the potential branch in question. The other tree used information gain (entropy), a measure of information content. The information content of a potential branch can be measured by the number of bits that would be required to encode it. Using entropy as a measure to determine the appropriate split leads branches to become increasingly less diverse, but it's important to recognize that the entropy and gini criteria can yield quite different results:

```
if __name__ == '__main__':

    np.random.seed(0)

    n_folds = 10
    verbose = True
    shuffle = False

    X, y, X_submission = load_data.load()

    if shuffle:
        idx = np.random.permutation(y.size)
        X = X[idx]
        y = y[idx]

    skf = list(StratifiedKFold(y, n_folds))

    clfs = [RandomForestClassifier(n_estimators=100, n_jobs=-1,
criterion='gini'),
            RandomForestClassifier(n_estimators=100, n_jobs=-1,
criterion='entropy'),
            ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
criterion='gini'),
            ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
criterion='entropy'),
            GradientBoostingClassifier(learning_rate=0.05,
subsample=0.5, max_depth=6, n_estimators=50)]

    print "Creating train and test sets for blending."

    dataset_blend_train = np.zeros((X.shape[0], len(clfs)))
    dataset_blend_test = np.zeros((X_submission.shape[0], len(clfs)))

    for j, clf in enumerate(clfs):
```

```
        print j, clf
        dataset_blend_test_j = np.zeros((X_submission.shape[0],
len(skf)))
        for i, (train, test) in enumerate(skf):
            print "Fold", i
            X_train = X[train]
            y_train = y[train]
            X_test = X[test]
            y_test = y[test]
            clf.fit(X_train, y_train)
            y_submission = clf.predict_proba(X_test)[:,1]
            dataset_blend_train[test, j] = y_submission
            dataset_blend_test_j[:, i] =
clf.predict_proba(X_submission)[:,1]
        dataset_blend_test[:,j] = dataset_blend_test_j.mean(1)

    print
    print "Blending."
    clf = LogisticRegression()
    clf.fit(dataset_blend_train, y)
    y_submission = clf.predict_proba(dataset_blend_test)[:,1]

    print "Linear stretch of predictions to [0,1]"
    y_submission = (y_submission - y_submission.min()) /
(y_submission.max() - y_submission.min())

    print "Saving Results."
    np.savetxt(fname='test.csv', X=y_submission, fmt='%0.9f')
```

When we try running this submission on the private leaderboard, we find ourselves in a rather impressive 12th place (out of 699 competitors)! Naturally, we can't draw too many conclusions from a competition that we entered after completion, but, given the simplicity of the code, this is still a rather impressive result!

Applying ensembles in practice

One particularly important quality to be mindful of while applying ensemble methods is that your goal is to tune the performance of the ensemble rather than of the models that comprise it. Your approach should therefore be largely focused on building a strong ensemble performance score, rather than the strongest set of individual model performances.

The amount of attention that you pay to the models within your ensemble will vary. With an arrangement of differently configured or initialized models of a single type (for example, a random forest), it is sensible to focus almost entirely on the performance of the ensemble and metaparameters that shape it.

For more challenging problems, we frequently need to pay closer attention to the individual models within our ensemble. This is most obviously true when we're trying to create smaller ensembles for more challenging problems, but to build a truly excellent ensemble, it is often necessary to be considerate of the parameters and algorithms underlying the structure that you've built.

With this said, you'll always be looking at the performance of the ensemble as well as the performance of models within the set. You'll be inspecting the results of your models to try and work out *what each model did well*. You'll also be looking for the less obvious factors that affect ensemble performance, most notably the correlation of model predictions. It's generally recognized that a more effective ensemble will tend to contain performant but uncorrelated components.

To understand this claim, consider techniques such as correlation measures and PCA that we can use to measure the amount of information content present in dataset variables. In the same way, we can use Pearson's correlation coefficient against the predictions output by each of our models to understand the relationship between performance and correlation for each model.

Taking us back to stacking ensembles specifically, our ensemble's models are outputting metafeatures that are then used as inputs to a next-layer model. Just as we would vet the features used by a more conventional neural network, we want to ensure that the features output by our ensemble's components work well as a dataset. The calculation of the Pearson correlation coefficient across model outputs and use of the results in model selection is an excellent place to start in this regard.

When we deal with single-model problems, we almost always have to spend some time inspecting the problem and identifying an appropriate learning algorithm. If we're faced with a two-class classification problem with a moderate amount of features (*10's*) and labeled training cases, we might select a logistic regression, an SVM, or some other appropriate algorithm for the context. Different approaches will apply to different problems and through trial and error, parallel testing, and experience (both personal and posted online!), you will identify the appropriate approach for a specific objective given specific input data.

A similar logic applies to ensemble creation. Rather than identifying a single appropriate model, the challenge is to identify combinations of models that effectively describe different elements of an input dataset in such a way that the dataset as a whole is adequately described. By understanding the strengths and weaknesses of your component models as well as by exploring and visualizing your dataset, you'll be able to draw conclusions about how to develop your ensemble effectively through multiple iterations.

Ultimately, at this level, data science is a field with a great many techniques at hand. The best practitioners are able to apply their knowledge of their own algorithms and options to develop very effective solutions over many iterations.

These solutions involve the knowledge of algorithms and interaction of model combinations, model parameter adjustments, dataset translations, and ensemble manipulation. Just as importantly, they require an uninhibited and creative mindset.

One good example of this is the work of prominent Kaggle competitor, Alexander Guschin. Focusing on one specific example—the **Otto Product Classification** contest—can give us an idea as to the range of options available to a confident and creative data scientist.

Most model development processes begin with a period in which you throw different solutions at the problem, attempting to find the tricks underlying the data and figuring out what works. Settling on a stacking model, Alexander set about building metafeatures. While we looked at XGBoost as an ensemble in its own right, in this case it was used as a component to the stacking ensemble in order to generate some of the metafeatures to be used by the final model. Neural networks were used in addition to the gradient boosted trees as both algorithms tend to produce good results.

To add some contrast to the mixture, Alexander added a KNN implementation, specifically because the results (and therefore the metaparameters) generated by a KNN tend to differ significantly from the models already included. This approach of picking up components whose outputs tend to differ is crucial in creating an effective stacking ensemble (and to most ensemble types).

To further develop this model, Alexander added some custom elements to the second layer of his model. While combining the XGBoost and neural network predictions, he also added bagging at this layer. At this point, most of the techniques that we've discussed in this chapter have shown up in some part of this model. In addition to the model development, some feature engineering (in particular, the use of TF-IDF on half of the training and test data) and the use of plotting techniques to identify class differentiation were used throughout.

A truly mature model that can tackle the most significant data science challenges is one that combines the techniques we've seen throughout this book, created using a solid understanding of the underlying algorithms and the possibilities for how these techniques can interact with each other.

This book so far has taught many of the fundamentals—the base of practical knowledge—that a practitioner has to collect. It has used many examples and an increasing amount of real-world cases to demonstrate how a broad base of knowledge becomes increasingly powerful in letting you develop effective solutions to difficult problems.

What's required of you as a data scientist is to first apply this broad set of techniques to develop an experience of how they can perform and what they could do for you. Then it is up to you to develop that creativity and experimental mindset that distinguishes some of the best data scientists.

Using models in dynamic applications

We've spent this chapter discussing the use of techniques to manage model performance under conditions that might be seen as ideal; specifically, conditions in which all of the data is available ahead of time so that a model can be trained on all data. These assumptions are frequently valid in research contexts or when dealing with one-time problems, but in many contexts they are unsafe assumptions. The range of unsafe contexts goes beyond the cases where the data is simply unavailable, such as data science contests where a held-out dataset is used to establish the final leaderboard.

Returning to a subject from earlier in this chapter, you'll recall the Pragmatic Chaos algorithm, which won the Netflix prize? By the time Netflix came to assessing the algorithm for implementation, both the business context and requirements had shifted so dramatically that the minimal accuracy gains provided by that algorithm didn't justify implementation costs. The $1M algorithm was redundant and was never implemented in production! The point to take from this example is that in commercial contexts, it is critical for our models to have as much adaptability as we can provide.

The really challenging applications of machine learning algorithms, in which our existing run once methodologies become less valuable, are ones where real data changes occur across time (or other dimensions). In these contexts, one knows that a substantial data change will occur and that existing models cannot be easily trained to adapt to this data change. At that point, new techniques are needed as well as new information.

To adapt and gather this information, we need to become better able to predict the ways in which data change is liable to occur. With this information, our model building and the content of our ensembles can start to change in order to cover the most likely data change scenarios that we see ahead. This adaptation lets us pre-empt data change and reduce the adjustment time required. As we'll see later in this chapter, in real-world applications any reduction in the time it takes us to pivot based on data change is valuable.

In the next section, we'll be looking at tools that we can use to make our models more robust to changing data. We'll discuss the means by which we can maintain a broad set of model options, simultaneously accommodating one or multiple data change scenarios, without reducing the performance of our models.

Understanding model robustness

It's important to understand exactly what the problem is here and how and when it is presented. This involves defining two things; the first being robustness as it applies to machine learning algorithms. The second, of course, is data change. Some of the content in the first part of this section is at an introductory level, but experienced data scientists may still find value in reviewing the section!

In academic terms, the robustness of a machine learning algorithm is the property that characterizes how effective your algorithm is while being applied to a dataset other than the dataset on which it was trained.
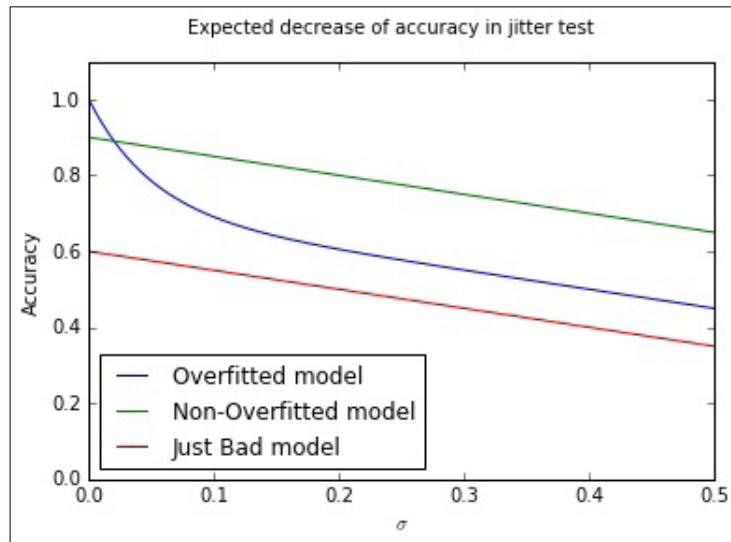
Robustness testing is a core part of machine learning methodology in any context. The importance of validation techniques such as k-fold cross-validation and the use of tests when developing models for even the simplest contexts is a consequence of machine learning algorithm vulnerability to data change.

Most datasets contain both a signal and noise. Noise may be predictable (and thus more easily managed) or it may be stochastic and difficult to treat. A dataset may contain more or less noise. Typically, datasets with more or less predictable noise are harder to train and test against the same datasets with this noise removed (which can be easily tested).

When one has trained a model on a given dataset, it is almost inevitable that this model has learned based on both the signal and noise. The concept of overfitting is generally used to describe a model that has fit so well to a given dataset that it has learned to predict based on both the signal and noise, rendering it less powerful against other samples than a model with a less exact fit.

Part of the goal of training a model is to reduce the impact of any local noise on learning as much as possible. The purpose of validation techniques that hold out a set of data to test is to ensure that any learning of noise during training happens only on noise that is local to the training set. The difference between training and test error can be used to understand the degree of overfitting between model implementations.

We've applied cross-validation in *Chapter 1, Unsupervised Machine Learning*. Another useful means of testing models for the overfitting is to directly add random noise in the form of jitter to the training dataset. This technique was introduced via a Kaggle notebook in October 2015 by Alexander Minushkin and offers a very interesting test. The concept is simple; by adding jitter and looking at the accuracy of prediction on the training data, we can distinguish an overfitted model (whose training error will increase more quickly as we add jitter) from a well- or poorly-fitted model:

In this case, we're able to plot the results of a jitter test to easily identify whether a model has overfit. From a very strong initial position, an overfit model will typically rapidly decline in performance as small amounts of jitter are added. For better-fitting models, the loss in performance with added jitter is much reduced, with the degree of overfitting in a model being particularly obvious at low levels of added jitter (where a well-fit model will tend to outperform an overfit counterpart).

Let's look at how we implement a jitter test for overfitting. We use a familiar score, `accuracy_score`, defined as the proportion of class labels predicted correctly, as the basis for test scoring. Jitter is defined by simply adding random noise to the data (using `np.random.normal`) with the amount of noise defined by the configurable `scale` parameter:

```
from sklearn.metrics import accuracy_score

def jitter(X, scale):
    if scale > 0:
        return X + np.random.normal(0, scale, X.shape)
    return X

def jitter_test(classifier, X, y, metric_FUNC = accuracy_score, sigmas
= np.linspace(0, 0.5, 30), averaging_N = 5):
    out = []

    for s in sigmas:
        averageAccuracy = 0.0
        for x in range(averaging_N):
            averageAccuracy += metric_FUNC( y, classifier.
predict(jitter(X, s)))

        out.append( averageAccuracy/averaging_N)

    return (out, sigmas, np.trapz(out, sigmas))

allJT = {}
```

The `jitter_test` itself is defined as a wrapper to normal sklearn classification, given a classifier, training data, and a set of target labels. The classifier is then called to predict against a version of the data that first has the `jitter` operation called against it.

At this point, we'll begin creating a number of datasets to run our jitter test over. We'll use sklearn's `make_moons` dataset, commonly used as a dataset to visualize clustering and classification algorithm performance. This dataset is comprised of two classes whose data points create interleaving half-circles. By adding varying amounts of noise to `make_moons` and using differing amounts of samples, we can create a range of example cases to run our jitter test against:

```
import sklearn
import sklearn.datasets

import warnings
```

```
warnings.filterwarnings("ignore", category=DeprecationWarning)

Xs = []
ys = []

#low noise, plenty of samples, should be easy
X0, y0 = sklearn.datasets.make_moons(n_samples=1000, noise=.05)
Xs.append(X0)
ys.append(y0)

#more noise, plenty of samples
X1, y1 = sklearn.datasets.make_moons(n_samples=1000, noise=.3)
Xs.append(X1)
ys.append(y1)

#less noise, few samples
X2, y2 = sklearn.datasets.make_moons(n_samples=200, noise=.05)
Xs.append(X2)
ys.append(y2)

#more noise, less samples, should be hard
X3, y3 = sklearn.datasets.make_moons(n_samples=200, noise=.3)
Xs.append(X3)
ys.append(y3)
```

This done, we then create a `plotter` object that we'll use to show our models'
performance directly against the input data:

```
def plotter(model, X, Y, ax, npts=5000):

    xs = []
    ys = []
    cs = []
    for _ in range(npts):
        x0spr = max(X[:,0])-min(X[:,0])
        x1spr = max(X[:,1])-min(X[:,1])
        x = np.random.rand()*x0spr + min(X[:,0])
        y = np.random.rand()*x1spr + min(X[:,1])
        xs.append(x)
        ys.append(y)
        cs.append(model.predict([x,y]))
    ax.scatter(xs,ys,c=list(map(lambda x:'lightgrey' if x==0 else
'black', cs)), alpha=.35)
    ax.hold(True)
```

```
        ax.scatter(X[:,0],X[:,1],
                    c=list(map(lambda x:'r' if x else 'lime',Y)),
                    linewidth=0,s=25,alpha=1)
        ax.set_xlim([min(X[:,0]), max(X[:,0])])
        ax.set_ylim([min(X[:,1]), max(X[:,1])])
        return
```

We'll use an SVM classifier as the base model for our jitter tests:

```
import sklearn.svm
classifier = sklearn.svm.SVC()


allJT[str(classifier)] = list()


fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(11,13))
i=0
for X,y in zip(Xs,ys):
    classifier.fit(X,y)
    plotter(classifier,X,y,ax=axes[i//2,i%2])
    allJT[str(classifier)].append (jitter_test(classifier, X, y))
    i += 1
plt.show()
```

The jitter test provides an effective means of assessing model overfitting and performs comparably to cross-validation; indeed, Minushkin provides evidence that it can outperform cross-validation as a tool to measure model fit quality.

Both of these tools to mitigate the overfitting work well in contexts where your algorithm is either run over data on a one-off basis or where underlying trends don't vary substantially. This is true for the majority of single-dataset problems (such as most academic or web repository datasets) or data problems where the underlying trends change slowly.

However, there are many contexts where the data involved in modeling might change over time in one or several dimensions. This can occur because of change in the methods by which data is captured, usually because new instruments or techniques come into use. For instance, video data captured by commonly-available devices has improved substantially in resolution over the decade since 2005 and the quality (and size!) of such data has increased. Whether you're using the video frames themselves or instead the file size as a parameter, you'll observe noticeable shifts in the nature, quality, and distributions of features.

Alternatively, changes in dataset variables might be caused by differences in underlying trends. The classic data schema concept of measures and dimensions comes back into play here, as we can better understand how data change is affected by considering what dimensions influence our measurement.

The key example is time. Depending on context, many variables are subject to day-of-week, month-of-year, or seasonal variations. In many cases, a helpful option might be to parameterize these variables, (as we discussed in the preceding chapter, techniques such as one-hot encoding can help our algorithms learn to parse such trends) particularly if we're dealing with periodic trends that are easily predicted (for example, the impact of month-of-year on scarf sales in a given location) and easily modeled.

A more problematic type of time series trend is non-periodic change. As in the preceding video camera example, some types of time series trends change irrevocably and in ways that might not be trivial to predict. Telemetry from software tends to be influenced by the quality and functionality of the software build live at the time the telemetry was emitted. As builds change over time, the values sent in telemetry and the variables created from those values can change radically overnight in hard-to-predict ways.

Human behavior, a hugely important factor in many datasets, helpfully changes both periodically and non-periodically. People shop more around seasonal holidays, but also change their shopping habits permanently based on new societal or technological developments.

Some of the added complexity here comes not just from the fact that single variables and their distributions are affected by time series trends, but also from how relationships between relevant factors and their associated variables will change. The relationships between variables may change in quantifiable terms. One example is how, for humans, height and weight are two variables whose relationship varies between times and locations. The BMI feature, which we might use to track this relationship, shows differing distributions when sampled across periods of time or between locations.

Furthermore, variables can change in another serious way; namely, their importance to a performant modeling algorithm may vary over time! Some variables whose values are highly relevant in some periods of time will be less relevant in others. As an example, consider how climate and weather variables affect agriculture markets. For some crops and the companies dealing in them, these variables are fairly unimportant for much of the year. At the time of crop growth and harvest, however, they become fundamentally important. To make this more complex, the strength of these factors' importance is also tied to location (and local climate).

The challenge for modeling is clear. For models that are trained once and run again on new data, managing data change can present serious challenges. For models that are dynamically recomputed based on new input data, data change can still create problems as variable distributions and relationships change and available variables become more or less valuable in generating an effective solution.

Part of the key to successfully managing data change in your application of ML is to recognize the dimensions (and there are common culprits) where change is probable and liable to affect the distributions of your features, relationships, and feature importance, which a model will attempt to pick up on.

Once you have an understanding as to what the factors in your data are that are likely to influence overfitting, you're better positioned to develop a solution that manages these factors effectively.

This said, it will still seem hugely challenging to build a single model that can resolve any potential issues. The simple response to this is that if one faces serious data change issues, the solution probably isn't to try to solve for them with a single model! In the next section, we'll be looking at ensemble methods to provide a better answer.

Identifying modeling risk factors

While it is in many cases quite straightforward to identify which elements present a risk to your model over time, it can help to employ a structured process for identification. This section briefly describes some of the heuristics and techniques you can employ to screen your models for the risk of data change.

Most data scientists keep a data dictionary for datasets that are intended for general use or automated applications. This is especially likely to happen if the data or applications are complex, but keeping a data dictionary is generally good practice. Some of the most effective work you can do in identifying risk factors is to run through these features and tag them based on different risk types.

Some of the tags that I tend to use include the following:

- **Longitudinally variant**: Is this parameter liable to change over a long time due to longitudinal trends that many not be fully visible in the span of the training data that you have available? The most obvious example is the ecological seasons, which affect many areas of human behavior as well as the many things that depend on some more fundamental climatic variables. Other longitudinal trends include the financial year and the working month, but extend to include many other longitudinal trends relevant to your area of investigation. The life cycle of new iPhone models or the population flux of voles might be an important longitudinal factor depending on the nature of your work.

- **Slowly changing**: Is this categorical parameter likely to gain new values over time? This concept is borrowed from data warehousing best practices. A slowly changing dimension in the classical sense will gain new parameter codes (for example, as a new store opens or a new case is identified). These can throw your model entirely if not managed properly or if they appear in sufficient number. Another impact of slowly changing data, which can be more problematic to handle, is that it can begin to affect the distribution of your features. This can have a substantial impact on the effectiveness of your model.

- **Key parameter**: A combination of data value monitoring and recalculation of decision boundaries/regression equations will often handle a certain amount of slowly changing data and seasonal variance well, but consider taking action should you see an unexpectedly large amount of new cases or case types, especially when they affect variables depended on heavily by your model. For this reason, also make sure that you know which variables are most relied upon by your solution!

The process of tagging in this way is helpful (not least as an export of your own memory) mostly because it helps you to do the following:

- Organize your expectations and develop a kind of checklist for your development of monitoring readiness. If you aren't able to keep track of at least your longitudinally variant and slowly changing parameter change, you are effectively blind to any output from your model besides changes in the parameters that it favors when recomputed and its (likely slowly declining) performance measure.

- Investigate mitigation (for example, improved normalization or extra parameters that codify those dimensions in which your data is variant). In many ways, mitigation and the addition of parameters is the best solution you can tap to handle data change.

- Set up robustness testing using constructed datasets, where your risk features are deliberately varied to simulate data change. Stress-test your model under these conditions and find out exactly how much variance it'll tolerate. With this information, you can easily set yourself up to use your monitoring values as an early alert system; once data change exceeds a certain safe threshold, you know how much degradation to expect in the model performance.

Strategies to managing model robustness

We've discussed a number of effective ensemble techniques that allow us to balance the twin needs for performant and robust models. However, throughout our exposition and use of these techniques, we had to decide how and when we would reduce our model's performance to improve robustness.

Indeed, a common theme in this chapter has been how to balance the conflicting objectives of creating an effective, performant model, without making this model too inflexible to respond to data change. Many of the solutions that we've seen so far have required that we trade-off one outcome against the other, which is less than ideal.

At this point, it's worth our taking a slightly wider view of our options and drawing from complimentary techniques. The need for robust, performant statistical models within evolving business landscapes is neither new nor untreated; fields such as credit risk modeling have a long history of applied statistical modeling in changing domains and have developed effective decision management methodologies in order to succeed. Data scientists can turn some of these established techniques to our own benefit via using them to help organize our own models.

One effective methodology is **Champion/Challenger**, a test-centric approach that involves running multiple, parallel model configurations. In addition to the model whose outputs are applied (to direct business activities or inform reporting), champion/challenger approaches training one or more alternative model configurations.

By maintaining and monitoring multiple models, one can arrange to substitute the current model as and when an alternative outperforms it. This is usually done by maintaining a performance scoring process for all models and observing the results so that a manual decision call can be made about whether and when to switch to a challenger.

While the simplest implementation may involve switching to a challenger as soon as it outperforms the main model, this is rarely done as there are risks around specific challenger models being exposed to local minima (for example, the day-of-week or month-of-year local trends). It is normal to spend a significant period assessing a challenger model, particularly ahead of sensitive applications. In complex real cases, one may even want to do additional testing by providing a sample of treatment cases to a promising challenger to determine whether it generates significant lift over the champion.

There is scope for some creativity beyond simple, "replace the challenger" succession rules. Voting-based approaches are quite common, where a top subset of the trained ensembles provides scores on a case-by-case basis and those scores treated as (weighted or unweighted) votes. Another approach involves using a **Borda count**, a voting system where each voter ranks the candidate solutions in order of preference. In the context of ensembling, one would typically assign each individual model's prediction a point value equal to its inverse rank (keeping each model separate!). Then one can combine these votes (usually experimenting with a range of different weightings) to generate a result.

Voting can perform fairly well with a larger number of models but is dependent on the specific modeling context and factors like the similarity of the different voters. As we discussed earlier in this chapter, it's critical to use tests such as Pearson's correlation coefficient to ensure that your model set is both performant and uncorrelated.

One may find that particular classes of input data (users, say, with specific segmentation tags) are more effectively treated by a given challenger and may implement a case routing system where multiple champions deal with different user subgroups. This approach overlaps somewhat with the benefits of boosting ensembles, but can help in production circumstances by separating concerns. However, maintaining multiple champions will increase the monitoring and oversight burden for your data team, so this option is best avoided if not entirely necessary.

A major concern to address is how we go about scoring our models, not least because there are immediate practical challenges. In particular, it is hard to compare multiple models in real contexts, given that class labels (to guide correctness) typically aren't available. In predictive contexts, this problem is compounded by the fact that the champion model's predictions are typically used to take actions that alter predicted events. This activity makes it very difficult to make assertions about how a challenger model's predictions would've performed; by taking action based on our champion's predictions, we're unable to confirm the results of our models!

The most common implementation process is to provide each challenger model with a statistically viable sample of the input data and then compare the lift from each approach. This approach inherently limits the number of challengers that one can support for some modeling problems. Another option is to leave just one statistically viable sample out of any treatment activity and use it to create a single regression test. This test is applied to the entire set of champion and challenger models, providing a meaningful basis for comparison.

The downside to this approach is that the change to a more effective model will always trail the data change by however long it takes to generate correct class labels for the test cases. While in many cases this isn't crippling (the champion model remains in place for the period it takes to generate accurate models), it can present problems in contexts where underlying conditions change rapidly compared to the training time for models.

> It's worth making one brief comment on the relationship between model training time and data change frequency. It isn't always clearly stated as such, but the typical goal in applied machine learning contexts is to reduce the factor of training time to data change frequency to the smallest value possible. To take the worst case, if the length of time it takes to train a model is longer than the length of time that model will be accurate for (and the ratio is equal to or greater than one), your model will never generate current results that can directly drive current actions. In general, a high ratio should prompt review and adjustment activities (either an investigation into whether faster score delivery at lower confidence delivers more value or adjustment to the rate at which controllable environment variables change).
>
> The smaller this ratio becomes, the more leeway your team has to apply your model's outputs to drive actions and generate value. Depending on how variant and quantifiable this ratio is for your modeling context, it can be a useful concept to promote within your organization as a health measure for your automated modeling solution.

These alternative models may simply be the next best-performing ensemble configurations; they may be older models, kept around for observation. In sophisticated operations, some challengers are configured to handle different *what-if* scenarios (for example, *what if the temperature in this region is 2 C below expectations* or *what if sales are significantly below expectations*). These models may have been trained on the same data as the main model or on deliberately skewed or prepared data that simulates the what-if scenario.

More challengers tend to be better (providing improved robustness and performance), provided that the challengers are not all minute variations on the same theme. Challenger models also provide a safe venue for innovation and testing, while observing effective challengers can provide useful insights into how robust your champion ensemble is likely to be to a range of possible environmental changes.

The techniques that you've learned to apply in this section have provided us with the tools to apply our existing toolkit of models to real applications in evolving environments. This chapter also discussed complications that can arise when applying ML models to production; data change, between samples or across dimensions, will cause our models to become increasingly ineffective. By thoroughly unpacking the concept of data change, we became better able to characterize this risk and recognize where and how it might present itself.

The remainder of the chapter was dedicated to techniques that provide improved model robustness. We discussed how to identify model degradation risk by looking at the underlying data and discussed some helpful heuristics to this end. We drew from existing decision management methods to learn about and use Champion/Challenger, a well-regarded process with a long history in contexts including applied machine learning. Champion/Challenger helps us organize and test multiple models in healthy competition. In conjunction with effective performance monitoring, a proactive tactical plan for model substitution will give you faster and more controllable management of the model life cycle and quality, all the while providing a wealth of valuable operational insights.

Further reading

Perhaps the most wide-ranging and informative tour of Ensembles and ensemble types is provided by the Kaggle competitor, Triskelion, at `http://mlwave.com/kaggle-ensembling-guide/`.

For discussion of the Netflix Prize-winning model, Pragmatic Chaos, refer to `http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BellKor.pdf`. For an explanation by Netflix on how changing business contexts rendered that $1M-model redundant, refer to the Netflix Tech blog at `http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`.

For a walkthrough on applying random forest ensembles to commercial contexts, with plenty of space given to all-important diagnostic charts and reasoning, consider Arshavir Blackwell's blog at `https://citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics/`.

For further information on random forests specifically, I find the scikit-learn documentation helpful: `http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`.

A great introduction to gradient-boosted trees is provided within the XGBoost documentation at `http://xgboost.readthedocs.io/en/latest/model.html`.

For a write-up of Alexander Guschin's entry to the Otto Product Classification challenge, refer to the No Free Hunch blog: `http://blog.kaggle.com/2015/06/09/otto-product-classification-winners-interview-2nd-place-alexander-guschin/`.

Alexander Minushkin's Jitter test for overfitting is described at `https://www.kaggle.com/miniushkin/introducing-kaggle-scripts/jitter-test-for-overfitting-notebook`.

Summary

In this chapter, we covered a lot of ground. We began by introducing ensembles, some of the most powerful and popular techniques in competitive machine learning contexts. We covered both the theory and code needed to apply ensembles to our machine learning projects, using a combination of expert knowledge and practical examples.

In addition, this chapter also dedicates a section to discussing the unique considerations that arise when you run models for weeks and months at a time. We discussed what data change can mean, how to identify it, and how to think about guarding against it. We gave specific consideration to the question of how to create sets of models running in parallel, which you can switch between based on seasonal change or performance drift in your model set.

During our review of these techniques, we spent significant time with real-world examples with the specific aim of learning more about the creative mindset and broad range of knowledge required of the best data scientists.

The techniques throughout this book have led up to a point that, armed with technical knowledge, code to reapply, and an understanding of the possibilities, you are truly able to take on any data modeling challenge.

9
Additional Python Machine Learning Tools

Over the course of the eight preceding chapters, we have examined and applied a range of techniques that help us enrich and model data for many applications.

We approached the content in these chapters using a combination of Python libraries, particularly NumPy and Theano, while the other libraries were drawn upon as and when we needed to access specific algorithms. We did not spend a great deal of time discussing what other options existed in terms of tools, what the unique differentiators of these tools were, or why we might be interested.

The primary goal of this final chapter is to highlight some other key libraries and frameworks that are available to you to use. These tools streamline and simplify the process of creating and applying models. This chapter presents these tools, demonstrates their application, and provides extensive advice regarding *Further reading*.

A major contributor to succeed in solving data science challenges and being successful as a data scientist is having a good understanding of the latest developments in algorithms and libraries. As professionals, data scientists tend to be highly dependent on the quality of the data they use, but it is also very important to have the best tools available.

In this chapter, we will review some of the best in the recent tools available to data scientists, identifying the benefits they offer, and discussing how to apply them alongside tools and techniques discussed earlier in this book within a consistent working process.

Alternative development tools

Over the last couple of years, a number of new machine learning frameworks have emerged that offer advantages in terms of workflow. Usually these frameworks are highly focused on a specific use case or objective. This makes them very useful, perhaps even must-have tools, but it also means that you may need to use multiple workflow improvement libraries.

With an ever-growing set of new Python ML projects being lit up to address specific workflow challenges, it's worth discussing two libraries that add to our existing workflow and which accelerate or improve the work we've done in the preceding chapters. In this chapter, we'll be introducing **Lasagne** and **TensorFlow**, discussing the code and capabilities of each library and identifying why each framework is worth considering as a part of your toolset.

Introduction to Lasagne

Let's face it; sometimes creating models in Python takes longer than we'd like. However, they can be efficient for models that are more complex and offer big benefits (such as GPU acceleration and configurability) libraries similar to Theano can be relatively complex to use when working on simple cases. This is unfortunate because we often want to work with simple models, for instance, when we're setting up benchmarks.

Lasagne is a library developed by a team of deep learning and music data mining researchers to work as an interface to Theano. It is designed specifically to nail a particular goal—to allow for fast and efficient prototyping of new models.

This focus dictated how Lasagne was created, to call Theano functions and return Theano expressions or `numpy` data types, in a much less complex and more easily understood manner than the same operations written in native Theano code.

In this section, we'll take a look at the conceptual model underlying Lasagne, apply some Lasagne code, and understand what the library adds to our existing practices.

Getting to know Lasagne

Lasagne operates using the concept of layers, a familiar concept in machine learning. A layer is a set of neurons and operating rules that will take an input and generate a score, label, or other transformations. Neural networks generally function as a set of layers that feed input data in at one end and push output values out at the other (though the ways in which this gets done vary broadly).

It has become very popular in deep learning contexts to start treating individual layers as first class citizens. Traditionally, in machine learning work, a network would be established from layers using only a few parameter specifications (such as node count, bias, and weight values).

In recent years, data scientists seeking that extra edge have begun to take increasing interest in the configuration of individual layers. Nowadays it is not unusual in advanced machine learning environments to see layers that contain sub-models and transformed inputs. Even features, nowadays, might skip layers as needed and new features may be added to layers partway through a model. As an example of some of this refinement, consider the convolutional neural network architectures employed by Google to solve image recognition challenges. These networks are extensively refined at a layer level to generate performance improvements.

It therefore makes sense that Lasagne treats layers as its basic model component. What Lasagne adds to the model creation process is the ability to stack different layers into a model quickly and intuitively. One may simply call a class within `lasagne.layers` to stack a class onto your model. The code for this is highly efficient and looks as follows:

```
l0 = lasagne.layers.InputLayer(shape=X.shape)

l1 = lasagne.layers.DenseLayer(
l0, num_units=10, nonlinearity=lasagne.nonlinearities.tanh)

l2 = lasagne.layers.DenseLayer(l1, num_units=N_CLASSES,
nonlinearity=lasagne.nonlinearities.softmax)
```

In three simple statements, we have created the basic structure of a network using simple and configurable functions.

This code creates a model using three layers. The layer `l0` calls the `InputLayer` class, acting as an input layer for our model. This layer translates our input dataset into a Theano tensor, based on the expected shape of the input (defined using the `shape` parameter).

The next layers, `l1` and `l2` are each fully connected (dense) layers. Layer `l2` is defined as an output layer, with a number of units equal to the number of classes, while `l1` uses the same `DenseLayer` class to create a hidden layer of `10` units.

In addition to configuration of the standard parameters (weights, biases, unit count and nonlinearity type) available to the `DenseLayer` class, it is possible to employ entirely different network types using different classes. Lasagne provides classes for a broad set of familiar layers, including dense, convolutional and pooling layers, recurrent layers, normalisation and noise layers, amongst others. There is, furthermore, a special-purpose layer class, which provides a range of additional functionality.

If something more bespoke than what these classes provide is needed, of course, the user can resort to defining their own layer type easily and use it in conjunction with other Lasagne classes. However, for a majority of prototyping and fast, iterative development contexts, this is a great amount of pre-prepared capability.

Lasagne provides a similarly succinct interface to define the loss calculation for a network:

```
true_output = T.ivector('true_output')
objective = lasagne.objectives.Objective(l2, loss_function=lasagne.
objectives.categorical_crossentropy)

loss = objective.get_loss(target=true_output)
```

The `loss` function defined here is one of the many available functions, including squared error, hinge loss for binary and multi-class cases, and `crossentropy` functions. An accuracy scoring function for validation is also provided.

With these two components, a `loss` function and a network architecture, we again have everything we need to train a network. To do this, we need to write a little more code:

```
all_params = lasagne.layers.get_all_params(l2)
updates = lasagne.updates.sgd(loss, all_params, learning_rate=1)
train = theano.function([l0.input_var, true_output], loss,
updates=updates)

get_output = theano.function([l0.input_var], net_output)

for n in xrange(100):
 train(X, y)
```

This code leverages the `theano` functionality to train our example network, using our `loss` function, to iteratively train to classify a given set of input data.

Introduction to TensorFlow

When we reviewed Google's take on the **convolutional neural network** (**CNN**) in *Chapter 4*, *Convolutional Neural Networks*, we found a convoluted, many-layered beast. The question of how to create and monitor such networks only became more important as the network scales in layer count and complexity to attack challenges that are more complex.

To address this challenge, the Machine Intelligence research organisation at Google developed and distributed a library named TensorFlow, which exists to enable easier refinement and modeling of very involved machine learning models.

TensorFlow does this by providing two main benefits; a clear and simple programming interface (in this case, a Python API) onto familiar structures (such as NumPy objects), and powerful diagnostic and graph visualisation tools, such as **TensorBoard**, to enable informed tuning of a data architecture.

Getting to know TensorFlow

TensorFlow enables a data scientist to design data transformation operations as a flow across a computation graph. This graph can be extended and modified, while individual nodes can be tuned extensively, enabling detailed refinements of individual layers or model components. The TensorFlow workflow typically involves two phases. The first of these is referred to as the construction phase, during which a graph is assembled.

During the construction phase, we can write code using the Python API for Tensorflow. Like Lasagne, TensorFlow offers a relatively simple interface to writing network layers, requiring simply that we specify weights and bias before creating our layers. The following example shows initial setting of weight and bias variables, before creating (using one line of code each) a convolutional layer and a simple max-pooling layer. Additionally, we use `tf.placeholder` to generate placeholder variables for our input data.

```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])

W = tf.Variable(tf.zeros([5, 5, 1, 32]))
b = tf.Variable(tf.zeros([32]))


h_conv = tf.nn.relu(conv2d(x_image, W) + b)
h_pool = max_pool_2x2(h_conv)
```

This structure can be extended to include a `softmax` output layer, just as we did with Lasagne.

```
W_out = tf.Variable(tf.zeros([1024,10]))
B_out = tf.Variable(tf.zeros([10]))

y = tf.nn.softmax(tf.matmul(h_conv, W_out) + b_out)
```

Again, we can see significant improvements in the iteration time over writing directly in Theano and Python libraries. Being written in C++, TensorFlow also provides performance gains over Python, providing advantages in execution time.

Next up, we need to train and evaluate our model. Here, we'll need to write a little code to define our `loss` function for training (cross entropy, in this case), an `accuracy` function for validation and an optimisation method (in this case, steepest gradient descent).

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
reduction_indices=[1]))

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_
entropy)

correct_prediction = tf.equal(tf.argmax(y_,1), tf.argmax(y_,1))

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Following this, we can simply begin running our model iteratively. This is all succinct and very straightforward:

```
sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob:
0.5})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```
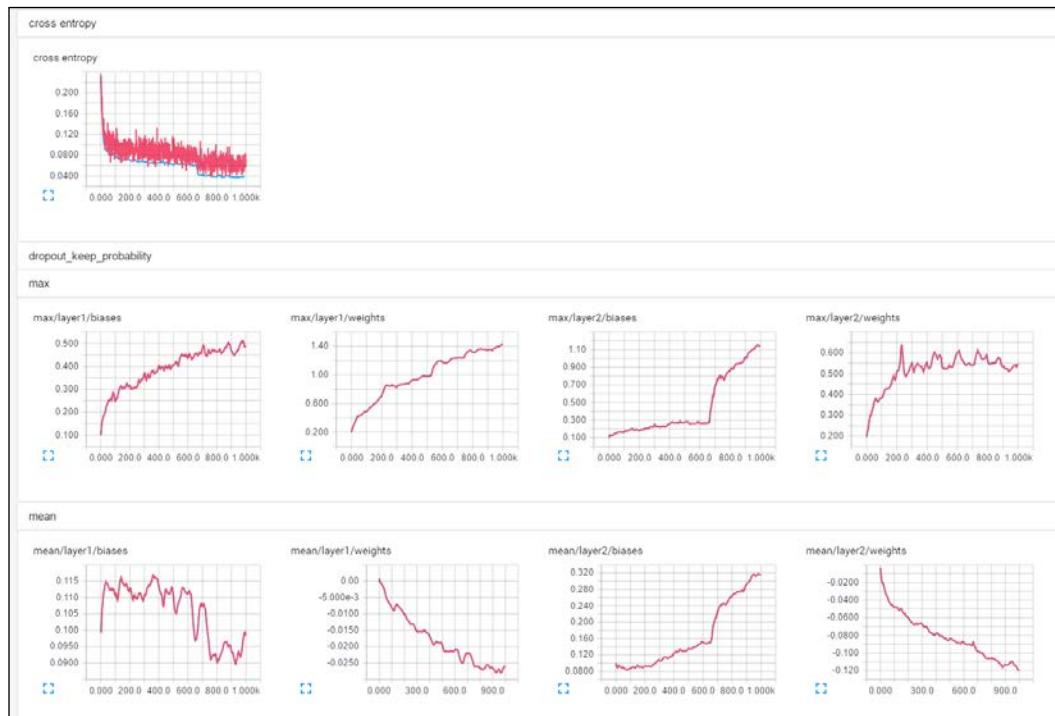
Using TensorFlow to iteratively improve our models

Even from the single example in the preceding section, we should be able to recognise what TensorFlow brings to the table. It offers a simple interface for the task of developing complex architectures and training methods, giving us easier access to the algorithms we've learnt about earlier in this book.
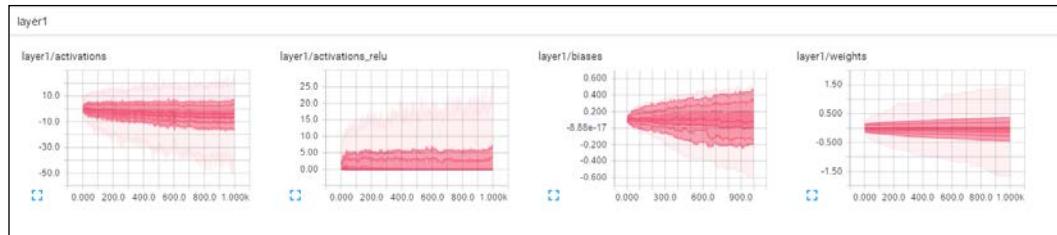
As we know, however, developing an initial model is only a small part of the model development process. We usually need to test and dissect our models repeatedly to improve their performance. However, this tends to be an area where our tools are less unified in a single library or technique, and the tests and monitoring solutions less consistent across models.

TensorFlow looks to solve the problem of how to get good insight into our models during iteration, in what it calls the execution phase of model development. During the execution phase, we can make use of tools provided by the TensorFlow team to explore and improve our models.
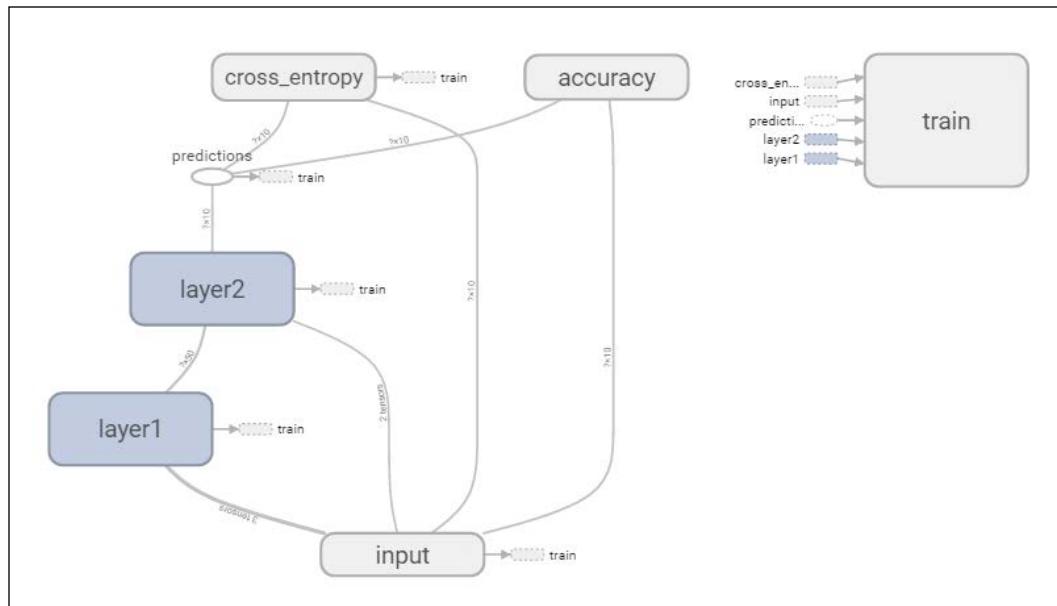
Perhaps the most important of these tools is TensorBoard, which provides an explorable, visual representation of the model we've built. TensorBoard provides several capabilities, including dashboards that show both basic model information (including performance measurements during each iteration for test and/or training).

In addition, TensorBoard dashboards provide lower-level information including plots of the range of values for weights, biases and activation values at every model layer; tremendously useful diagnostic information during iteration. The process of accessing this data is hassle-free and it is immediately useful.
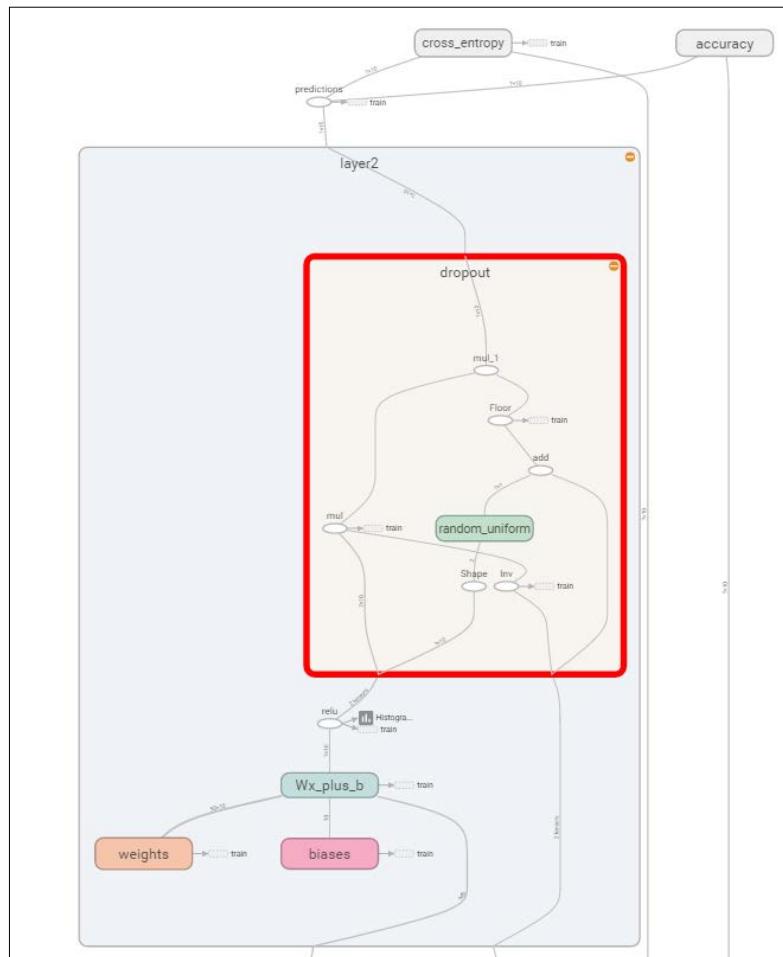

Further to this, TensorBoard provides a detailed graph of the tensor flow for a given model. The tensor is an n-dimensional array of data (in this case, of n-many features); it's what we tend to think of when we use the term *the input dataset*. The series of operations that is applied to a tensor is described as the tensor flow and in TensorFlow it's a fundamental concept, for a simple and compelling reason. When refining and debugging a machine learning model, what matters is having information about the model and its operations at even a low level.

TensorBoard graphs show the structure of a model in variable detail. From this initial view, it is possible to dig into each component of the model and into successive sub-elements. In this case, we are able to view the specific operations that take place within the dropout function of our second network layer. We can see what happens and identify what to tweak for our next iteration.

This level of transparency is unusual and can be very helpful when we want to tweak model components, especially when a model element or layer is underperforming (as we might see, for instance, from TensorBoard graphs showing layer metaparameter values or from network performance as a whole).

TensorBoards can be created from event logs and generated when TensorFlow is run. This makes the benefits of TensorBoards easily obtained during the course of everyday development using TensorFlow.

As of late April 2016, the DeepMind team joined the Google Brain team and a broad set of other researchers and developers in using TensorFlow. By making TensorFlow open source and freely available, Google is committing to continue supporting TensorFlow as a powerful tool for model development and refinement.

Knowing when to use these libraries

At one or two points in this chapter, we probably ran into the question of *Okay, so, why didn't you just teach us about this library to begin with?* It's fair to ask why we spent time digging around in Theano functions and other low-level information when this chapter presents perfectly good interfaces that make life easier.

Naturally, I advocate using the best tools available, especially for prototyping tasks where the value of the work is more in understanding the general ballpark you're in, or in identifying specific problem classes. It's worth recognising the three reasons for not presenting content earlier in this book using either of these libraries.

The first reason is that these tools will only get you so far. They can do a lot, agreed, so depending on the domain and the nature of that domain's problems, some data scientists may be able to rely on them for the majority of deep learning needs. Beyond a certain level of performance and problem complexity, of course, you need to understand what is needed to construct a model in Theano, create your own scoring function from scratch or leverage the other techniques described in this book.

Another part of the decision to focus on teaching lower-level implementation is about the developing maturity of the technologies involved. At this point, Lasagne and TensorFlow are definitely worth discussing and recommending to you. Prior to this, when the majority of the book was written, the risk around discussing the libraries in this chapter was greater. There are many projects based on Theano (some of the more prominent frameworks which weren't discussed in this chapter are **Keras**, **Blocks** and **Pylearn2**)

Even now, it's entirely possible that different libraries and tools will be the subject of discussion or the default working environment in a year or two years' time. This field moves extremely fast, largely due to the influence of key companies and research groups who have to keep building new tools as the old ones reach their useful limits… or it just becomes clear how to do things better.

The other reason to dig in at a lower level, honestly, is that this is an involved book. It sets theory alongside code and uses the code to teach the theory. Abstracting away how the algorithms work and simply discussing how to apply them to crack a particular example can be tempting. The tools discussed in this chapter enable practitioners to get very good scores on some problems without ever understanding the functions that are being called. My opinion is that this is not a very good way to train a data scientist.

If you're going to operate on subtle and difficult data problems, you need to be able to modify and define your own algorithm. You need to understand how to choose an appropriate solution. To do these things, you need the details provided in this book and even more very specific information that I haven't provided, due to the limitations of (page) space and time. At that point, you can apply deep learning algorithms flexibly and knowledgeably.

Similarly, it's important to recognise what these tools do well, or less well. At present, Lasagne fits very well within that use-case where a new model is being developed for benchmarking or early passes, where the priority should be on iteration speed and getting results.

TensorFlow, meanwhile, fits later into the development lifespan of a model. When the easy gains disappear and it's necessary to spend a lot of time debugging and improving a model, the relatively quick iterations of TensorFlow are a definite plus, but it's the diagnostic tools provided by TensorBoard that present an overwhelming value-add.

There is, therefore, a place for both libraries in your toolset. Depending on the nature of the problem at hand, these libraries and more will prove to be valuable assets.

Further reading

The Lasagne User Guide is thorough and worth reading. Find it at `http://lasagne.readthedocs.io/en/latest/index.html`.

Similarly, find the TensorFlow tutorials at `https://www.tensorflow.org/versions/r0.9/get_started/index.html`.

Summary

In this final chapter, we moved some distance from our previous discussions of algorithms, configuration and diagnosis to consider tools that improve our experience when implementing deep learning algorithms.

We discovered the advantages to using Lasagne, an interface to Theano designed to accelerate and simplify early prototyping of our models. Meanwhile, we examined TensorFlow, the library developed by Google to aid Deep Learning model adjustment and optimization. TensorFlow offers us a remarkable amount of visibility of model performance, at minimal effort, and makes the task of diagnosing and debugging a complex, deep model structure much less challenging.

Both tools have their own place in our processes, with each being appropriate for a particular set of problems.

Over the course of this book as a whole, we have walked through and reviewed a broad set of advanced machine learning techniques. We went from a position where we understood some fundamental algorithms and concepts, to having confident use of a very current, powerful and sought-after toolset.

Beyond the techniques, though, this book attempts to teach one further concept, one that's much harder to teach and to learn, but which underpins the best performance in machine learning.

The field of machine learning is moving very fast. This pace is visible in new and improved scores that are posted almost every week in academic journals or industry white papers. It's visible in how training examples like MNIST have moved quickly from being seen as meaningful challenges to being *toy problems*, the deep learning version of the Iris dataset. Meanwhile, the field moves on to the next big challenge; CIFAR-10, CIFAR-100.

At the same time, the field moves cyclically. Concepts introduced by academics like Yann LeCun in the 80's are in resurgence as computing architectures and resource growth make their use more viable over real data at scale. To use many of the most current techniques at their best limits, it's necessary to understand concepts that were defined decades ago, themselves defined on the back of other concepts defined still longer ago.

This book tries to balance these concerns. Understanding the cutting edge and the techniques that exist there is critical; understanding the concepts that'll define the new techniques or adjustments made in two or three years' time is equally important.

Most important of all, however, is that this book gives you an appreciation of how malleable these architectures and approaches can be. A concept consistently seen at the top end of data science practice is that the best solution to a specific problem is a problem-specific solution.

This is why top Kaggle contest winners perform extensive feature preparation and tweak their architectures. It's why TensorFlow was written to allow clear vision of granular properties of ones' architectures. Having the knowledge and the skills to tweak implementations or combine algorithms fluently is what it takes to have true mastery of machine learning techniques.

Through the many techniques and examples reviewed within this book, it is my hope that the ways of thinking about data problems and a confidence in manipulating and configuring these algorithms has been passed on to you as a practicing data scientist. The many recommended *Further reading* examples in this book are largely intended to further extend that knowledge and help you develop the skills taught in this book.

Beyond that, I wish you all the best of luck in your model building and configuration. I hope that you learn for yourself just how enjoyable and rewarding this field can be!

10

Chapter Code Requirements

This book's content leverages openly available data and code, including open source Python libraries and frameworks. While each chapter's example code is accompanied by a `README` file documenting all the libraries required to run the code provided in that chapter's accompanying scripts, the content of these files is collated here for your convenience.

It is recommended that you already have some libraries that are required for the earlier chapters when working with code from any later chapter. These requirements are identified using keywords. It is particularly important to set up the libraries mentioned in *Chapter 1*, *Unsupervised Machine Learning*, for any content provided later in the book. The requirements for every chapter are given in the following table:

Chapter Number	Requirements
1	• Python 3 (3.4 recommended) • sklearn (NumPy, SciPy) • matplotlib
2-4	• theano
5	• Semisup-learn
6	• **Natural Language Toolkit** (**NLTK**) • BeautifulSoup
7	• Twitter API account
8	• XGBoost
9	• Lasagne • TensorFlow

Biblography

This course is packaged keeping your journey in mind. It includes content from the following Packt products:

- *Python Machine Learning, Sebastian Raschka*
- *Designing Machine Learning Systems with Python, David Julian*
- *Advanced Machine Learning with Python, John Hearty*

Thank you for buying

Python: Deeper Insights into Machine Learning

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Please check `www.PacktPub.com` for information on our titles