Community Experience Distilled

# Mastering Machine Learning with scikit-learn

Apply effective learning algorithms to real-world problems using scikit-learn

Gavin Hackeling

[PACKT] open source* PUBLISHING
community experience distilled

# Mastering Machine Learning with scikit-learn

Apply effective learning algorithms to real-world problems using scikit-learn

**Gavin Hackeling**

[PACKT] PUBLISHING
open source
community experience distilled
BIRMINGHAM - MUMBAI

# Mastering Machine Learning with scikit-learn



First published: October 2014

Production reference: 1221014

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

ISBN 978-1-78398-836-5

www.packtpub.com

Cover image by Amy-Lee Winfield (abjure@outlook.com)

# Credits

**Author**
Gavin Hackeling

**Reviewers**
Fahad Arshad
Sarah Guido
Mikhail Korobov
Aman Madaan

**Acquisition Editor**
Meeta Rajani

**Content Development Editor**
Neeshma Ramakrishnan

**Technical Editor**
Faisal Siddiqui

**Copy Editors**
Roshni Banerjee
Adithi Shetty

**Project Coordinator**
Danuta Jones

**Proofreaders**
Simran Bhogal
Tarsonia Sanghera
Lindsey Thomas

**Indexer**
Monica Ajmera Mehta

**Graphics**
Sheetal Aute
Ronak Dhruv
Disha Haria

**Production Coordinator**
Kyle Albuquerque

**Cover Work**
Kyle Albuquerque

# About the Author

**Gavin Hackeling** develops machine learning services for large-scale documents and image classification at an advertising network in New York. He received his Master's degree from New York University's Interactive Telecommunications Program, and his Bachelor's degree from the University of North Carolina.

---

To Hallie, for her support, and Zipper, without whose contributions this book would have been completed in half the time.

---

# About the Reviewers

**Fahad Arshad** completed his PhD at Purdue University in the Department of Electrical and Computer Engineering. His research interests focus on developing algorithms for software testing, error detection, and failure diagnosis in distributed systems. He is particularly interested in data-driven analysis of computer systems. His work has appeared at top dependability conferences—DSN, ISSRE, ICAC, Middleware, and SRDS—and he has been awarded grants to attend DSN, ICAC, and ICNP. Fahad has also been an active contributor to security research while working as a cybersecurity engineer at NEEScomm IT. He has recently taken on a position as a systems engineer in the industry.

**Sarah Guido** is a data scientist at Reonomy, where she's helping build disruptive technology in the commercial real estate industry. She loves Python, machine learning, and the startup world. She is an accomplished conference speaker and an O'Reilly Media author, and is very involved in the Python community. Prior to joining Reonomy, Sarah earned a Master's degree from the University of Michigan School of Information.

**Mikhail Korobov** is a software developer at ScrapingHub Inc., where he works on web scraping, information extraction, natural language processing, machine learning, and web development tasks. He is an NLTK team member, Scrapy team member, and an author or contributor to many other open source projects.

---

I'd like to thank my wife, Aleksandra, for her support and patience and for the cookies.

---

**Aman Madaan** is currently pursuing his Master's in Computer Science and Engineering. His interests span across machine learning, information extraction, natural language processing, and distributed computing. More details about his skills, interests, and experience can be found at `http://www.amanmadaan.in`.

# www.PacktPub.com

## Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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

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


`http://PacktLib.PacktPub.com`

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

### Why subscribe?

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

### Free access for Packt account holders

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

# Table of Contents

# Preface

Recent years have seen the rise of machine learning, the study of software that learns from experience. While machine learning is a new discipline, it has found many applications. We rely on some of these applications daily; in some cases, their successes have already rendered them mundane. Many other applications have only recently been conceived, and hint at machine learning's potential.

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

This book is motivated by two goals:

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

# What this book covers

*Chapter 1*, *The Fundamentals of Machine Learning*, defines machine learning as the study and design of programs that improve their performance of a task by learning from experience. This definition guides the other chapters; in each chapter, we will examine a machine learning model, apply it to a task, and measure its performance.

*Chapter 2*, *Linear Regression*, discusses linear regression, a model that relates explanatory variables and model parameters to a continuous response variable. You will learn about cost functions, and use the normal equation to find the parameter values that produce the optimal model.

*Chapter 3*, *Feature Extraction and Preprocessing*, describes methods to represent text, images, and categorical variables as features that can be used in machine learning models.

*Chapter 4*, *From Linear Regression to Logistic Regression*, discusses generalizing linear regression to support classification tasks. We combine a model called logistic regression with some of the feature engineering techniques from the previous chapter to create a spam filter.

*Chapter 5*, *Nonlinear Classification and Regression with Decision Trees*, departs from linear models to discuss classification and regression with models called decision trees. We use an ensemble of decision trees to construct a banner advertisement blocker.

*Chapter 6*, *Clustering with K-Means*, introduces unsupervised learning. We examine the k-means algorithm, and combine it with logistic regression to create a semi-supervised photo classifier.

*Chapter 7*, *Dimensionality Reduction with PCA*, discusses another unsupervised learning task called dimensionality reduction. We use principal component analysis to visualize high-dimensional data and build a face recognizer.

*Chapter 8*, *The Perceptron*, describes an online, binary classifier called the perceptron. The limitations of the perceptron motivate the models described in the final chapters.

*Chapter 9*, *From the Perceptron to Support Vector Machines*, discusses efficient nonlinear classification and regression with support vector machines. We use support vector machines to recognize the characters in photographs of street signs.

*Chapter 10*, *From the Perceptron to Artificial Neural Networks*, introduces powerful nonlinear models for classification and regression called artificial neural networks. We build a network that can recognize handwritten digits.

# What you need for this book

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

# Who this book is for

This book is intended for software developers who have some experience with machine learning. scikit-learn's API is well-documented, but assumes that the reader understands how machine learning algorithms work and when it is appropriate to use them. This book does not attempt to reproduce the API's documentation. Instead, it describes how machine learning models work, how their parameters are learned, and how they can be evaluated. When practical, we will work through toy examples of the algorithms in detail to build the understanding required to apply them effectively.

# Conventions

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

In-line code is formatted as follows: "The `TfidfVectorizer` combines the `CountVectorizer` and the `TfidfTransformer`."

A block of code is indicated as follows:

```
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from sklearn.linear_model.logistic import LogisticRegression
>>> from sklearn.cross_validation import train_test_split
>>> df = pd.read_csv('sms/sms.csv')
>>> X_train_raw, X_test_raw, y_train, y_test = train_test_
split(df['message'], df['label'])
>>> vectorizer = TfidfVectorizer()
>>> X_train = vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
```

# Reader feedback

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

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `www.packtpub.com/authors`.

# Customer support

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

# Downloading the example code

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

# Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the errata submission form link, and entering the details of your errata. Once your errata has been 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. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

# Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you experience any problems with any aspect of this book, and we will do our best to address it.

# 1
# The Fundamentals of Machine Learning

In this chapter we will review the fundamental concepts in machine learning. We will discuss applications of machine learning algorithms, the supervised-unsupervised learning spectrum, uses of training and testing data, and model evaluation. Finally, we will introduce scikit-learn, and install the tools required in subsequent chapters.

Our imagination has long been captivated by visions of machines that can learn and imitate human intelligence. While visions of general artificial intelligence such as Arthur C. Clarke's HAL and Isaac Asimov's Sonny have yet to be realized, software programs that can acquire new knowledge and skills through experience are becoming increasingly common. We use such machine learning programs to discover new music that we enjoy, and to quickly find the exact shoes we want to purchase online. Machine learning programs allow us to dictate commands to our smartphones and allow our thermostats to set their own temperatures. Machine learning programs can decipher sloppily-written mailing addresses better than humans, and guard credit cards from fraud more vigilantly. From investigating new medicines to estimating the page views for versions of a headline, machine learning software is becoming central to many industries. Machine learning has even encroached on activities that have long been considered uniquely human, such as writing the sports column recapping the Duke basketball team's loss to UNC.

Machine learning is the design and study of software artifacts that use past experience to make future decisions; it is the study of programs that learn from data. The fundamental goal of machine learning is to *generalize*, or to induce an unknown rule from examples of the rule's application. The canonical example of machine learning is spam filtering. By observing thousands of emails that have been previously labeled as either spam or ham, spam filters learn to classify new messages.

Arthur Samuel, a computer scientist who pioneered the study of artificial intelligence, said that machine learning is "the study that gives computers the ability to learn without being explicitly programmed." Throughout the 1950s and 1960s, Samuel developed programs that played checkers. While the rules of checkers are simple, complex strategies are required to defeat skilled opponents. Samuel never explicitly programmed these strategies, but through the experience of playing thousands of games, the program learned complex behaviors that allowed it to beat many human opponents.

A popular quote from computer scientist Tom Mitchell defines machine learning more formally: "A program can be said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*." For example, assume that you have a collection of pictures. Each picture depicts either a dog or cat. A task could be sorting the pictures into separate collections of dog and cat photos. A program could learn to perform this task by observing pictures that have already been sorted, and it could evaluate its performance by calculating the percentage of correctly classified pictures.

We will use Mitchell's definition of machine learning to organize this chapter. First, we will discuss types of experience, including **supervised** learning and **unsupervised** learning. Next, we will discuss common tasks that can be performed by machine learning systems. Finally, we will discuss performance measures that can be used to assess machine learning systems.

# Learning from experience

Machine learning systems are often described as learning from experience either with or without supervision from humans. In supervised learning problems, a program predicts an output for an input by learning from pairs of labeled inputs and outputs; that is, the program learns from examples of the right answers. In unsupervised learning, a program does not learn from labeled data. Instead, it attempts to discover patterns in the data. For example, assume that you have collected data describing the heights and weights of people. An example of an unsupervised learning problem is dividing the data points into groups. A program might produce groups that correspond to men and women, or children and adults.

Now assume that the data is also labeled with the person's sex. An example of a supervised learning problem is inducing a rule to predict whether a person is male or female based on his or her height and weight. We will discuss algorithms and examples of supervised and unsupervised learning in the following chapters.

Supervised learning and unsupervised learning can be thought of as occupying opposite ends of a spectrum. Some types of problems, called **semi-supervised** learning problems, make use of both supervised and unsupervised data; these problems are located on the spectrum between supervised and unsupervised learning. An example of semi-supervised machine learning is reinforcement learning, in which a program receives feedback for its decisions, but the feedback may not be associated with a single decision. For example, a reinforcement learning program that learns to play a side-scrolling video game such as *Super Mario Bros.* may receive a reward when it completes a level or exceeds a certain score, and a punishment when it loses a life. However, this supervised feedback is not associated with specific decisions to run, avoid Goombas, or pick up fire flowers. While this book will discuss semi-supervised learning, we will focus primarily on supervised and unsupervised learning, as these categories include most the common machine learning problems. In the next sections, we will review supervised and unsupervised learning in more detail.

A supervised learning program learns from labeled examples of the outputs that should be produced for an input. There are many names for the output of a machine learning program. Several disciplines converge in machine learning, and many of those disciplines use their own terminology. In this book, we will refer to the output as the **response variable**. Other names for response variables include dependent variables, regressands, criterion variables, measured variables, responding variables, explained variables, outcome variables, experimental variables, labels, and output variables. Similarly, the input variables have several names. In this book, we will refer to the input variables as **features**, and the phenomena they measure as **explanatory variables**. Other names for explanatory variables include predictors, regressors, controlled variables, manipulated variables, and exposure variables. Response variables and explanatory variables may take real or discrete values.

The collection of examples that comprise supervised experience is called a **training set**. A collection of examples that is used to assess the performance of a program is called a **test set**. The response variable can be thought of as the answer to the question posed by the explanatory variables. Supervised learning problems learn from a collection of answers to different questions; that is, supervised learning programs are provided with the correct answers and must learn to respond correctly to unseen, but similar, questions.

# Machine learning tasks

Two of the most common supervised machine learning tasks are **classification** and **regression**. In classification tasks the program must learn to predict discrete values for the response variables from one or more explanatory variables. That is, the program must predict the most probable category, class, or label for new observations. Applications of classification include predicting whether a stock's price will rise or fall, or deciding if a news article belongs to the politics or leisure section. In regression problems the program must predict the value of a continuous response variable. Examples of regression problems include predicting the sales for a new product, or the salary for a job based on its description. Similar to classification, regression problems require supervised learning.

A common unsupervised learning task is to discover groups of related observations, called **clusters**, within the training data. This task, called **clustering** or cluster analysis, assigns observations to groups such that observations within groups are more similar to each other based on some similarity measure than they are to observations in other groups. Clustering is often used to explore a dataset. For example, given a collection of movie reviews, a clustering algorithm might discover sets of positive and negative reviews. The system will not be able to label the clusters as "positive" or "negative"; without supervision, it will only have knowledge that the grouped observations are similar to each other by some measure. A common application of clustering is discovering segments of customers within a market for a product. By understanding what attributes are common to particular groups of customers, marketers can decide what aspects of their campaigns need to be emphasized. Clustering is also used by Internet radio services; for example, given a collection of songs, a clustering algorithm might be able to group the songs according to their genres. Using different similarity measures, the same clustering algorithm might group the songs by their keys, or by the instruments they contain.

**Dimensionality reduction** is another common unsupervised learning task. Some problems may contain thousands or even millions of explanatory variables, which can be computationally costly to work with. Additionally, the program's ability to generalize may be reduced if some of the explanatory variables capture noise or are irrelevant to the underlying relationship. Dimensionality reduction is the process of discovering the explanatory variables that account for the greatest changes in the response variable. Dimensionality reduction can also be used to visualize data. It is easy to visualize a regression problem such as predicting the price of a home from its size; the size of the home can be plotted on the graph's $x$ axis, and the price of the home can be plotted on the $y$ axis. Similarly, it is easy to visualize the housing price regression problem when a second explanatory variable is added. The number of bathrooms in the house could be plotted on the z axis, for instance. A problem with thousands of explanatory variables, however, becomes impossible to visualize.

# Training data and test data

The observations in the training set comprise the experience that the algorithm uses to learn. In supervised learning problems, each observation consists of an observed response variable and one or more observed explanatory variables.

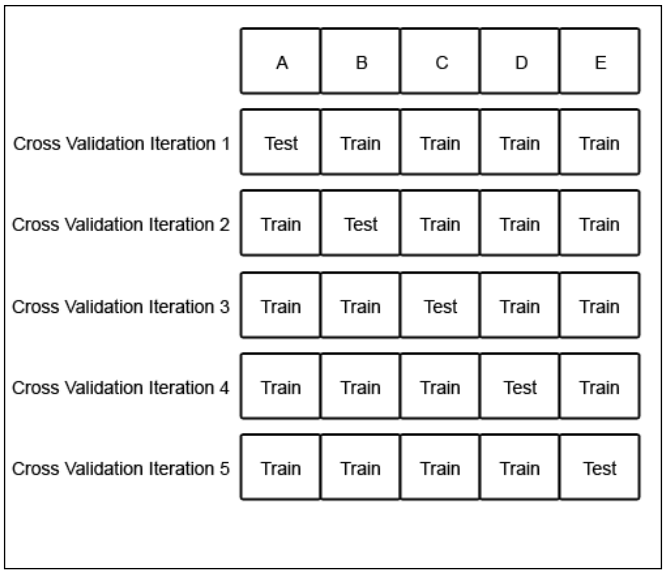
The test set is a similar collection of observations that is used to evaluate the performance of the model using some performance metric. It is important that no observations from the training set are included in the test set. If the test set does contain examples from the training set, it will be difficult to assess whether the algorithm has learned to generalize from the training set or has simply memorized it. A program that generalizes well will be able to effectively perform a task with new data. In contrast, a program that memorizes the training data by learning an overly complex model could predict the values of the response variable for the training set accurately, but will fail to predict the value of the response variable for new examples.

Memorizing the training set is called **over-fitting**. A program that memorizes its observations may not perform its task well, as it could memorize relations and structures that are noise or coincidence. Balancing memorization and generalization, or over-fitting and under-fitting, is a problem common to many machine learning algorithms. In later chapters we will discuss regularization, which can be applied to many models to reduce over-fitting.

In addition to the training and test data, a third set of observations, called a **validation** or **hold-out set**, is sometimes required. The validation set is used to tune variables called **hyperparameters**, which control how the model is learned. The program is still evaluated on the test set to provide an estimate of its performance in the real world; its performance on the validation set should not be used as an estimate of the model's real-world performance since the program has been tuned specifically to the validation data. It is common to partition a single set of supervised observations into training, validation, and test sets. There are no requirements for the sizes of the partitions, and they may vary according to the amount of data available. It is common to allocate 50 percent or more of the data to the training set, 25 percent to the test set, and the remainder to the validation set.

Some training sets may contain only a few hundred observations; others may include millions. Inexpensive storage, increased network connectivity, the ubiquity of sensor-packed smartphones, and shifting attitudes towards privacy have contributed to the contemporary state of big data, or training sets with millions or billions of examples. While this book will not work with datasets that require parallel processing on tens or hundreds of machines, the predictive power of many machine learning algorithms improves as the amount of training data increases. However, machine learning algorithms also follow the maxim "garbage in, garbage out." A student who studies for a test by reading a large, confusing textbook that contains many errors will likely not score better than a student who reads a short but well-written textbook. Similarly, an algorithm trained on a large collection of noisy, irrelevant, or incorrectly labeled data will not perform better than an algorithm trained on a smaller set of data that is more representative of problems in the real world.

Many supervised training sets are prepared manually, or by semi-automated processes. Creating a large collection of supervised data can be costly in some domains. Fortunately, several datasets are bundled with scikit-learn, allowing developers to focus on experimenting with models instead. During development, and particularly when training data is scarce, a practice called **cross-validation** can be used to train and validate an algorithm on the same data. In cross-validation, the training data is partitioned. The algorithm is trained using all but one of the partitions, and tested on the remaining partition. The partitions are then rotated several times so that the algorithm is trained and evaluated on all of the data. The following diagram depicts cross-validation with five partitions or **folds**:

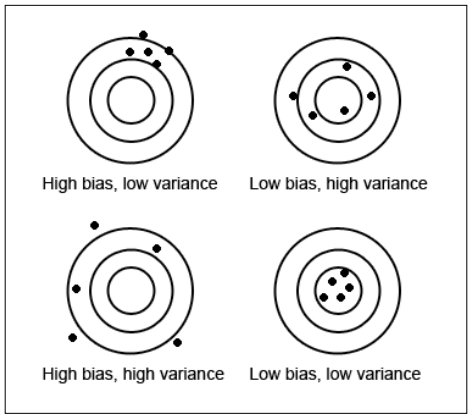


The original dataset is partitioned into five subsets of equal size, labeled **A** through **E**. Initially, the model is trained on partitions **B** through **E**, and tested on partition **A**. In the next iteration, the model is trained on partitions **A**, **C**, **D**, and **E**, and tested on partition **B**. The partitions are rotated until models have been trained and tested on all of the partitions. Cross-validation provides a more accurate estimate of the model's performance than testing a single partition of the data.

# Performance measures, bias, and variance

Many metrics can be used to measure whether or not a program is learning to perform its task more effectively. For supervised learning problems, many performance metrics measure the number of prediction errors. There are two fundamental causes of prediction error: a model's **bias** and its **variance**. Assume that you have many training sets that are all unique, but equally representative of the population. A model with a high bias will produce similar errors for an input regardless of the training set it was trained with; the model biases its own assumptions about the real relationship over the relationship demonstrated in the training data. A model with high variance, conversely, will produce different errors for an input depending on the training set that it was trained with. A model with high bias is inflexible, but a model with high variance may be so flexible that it models the noise in the training set. That is, a model with high variance over-fits the training data, while a model with high bias under-fits the training data. It can be helpful to visualize bias and variance as darts thrown at a dartboard. Each dart is analogous to a prediction from a different dataset. A model with high bias but low variance will throw darts that are far from the bull's eye, but tightly clustered. A model with high bias and high variance will throw darts all over the board; the darts are far from the bull's eye and each other.

A model with low bias and high variance will throw darts that are closer to the bull's eye, but poorly clustered. Finally, a model with low bias and low variance will throw darts that are tightly clustered around the bull's eye, as shown in the following diagram:




Ideally, a model will have both low bias and variance, but efforts to decrease one will frequently increase the other. This is known as the **bias-variance trade-off**. We will discuss the biases and variances of many of the models introduced in this book.

Unsupervised learning problems do not have an error signal to measure; instead, performance metrics for unsupervised learning problems measure some attributes of the structure discovered in the data.

Most performance measures can only be calculated for a specific type of task. Machine learning systems should be evaluated using performance measures that represent the costs associated with making errors in the real world. While this may seem obvious, the following example describes the use of a performance measure that is appropriate for the task in general but not for its specific application.

Consider a classification task in which a machine learning system observes tumors and must predict whether these tumors are malignant or benign. **Accuracy**, or the fraction of instances that were classified correctly, is an intuitive measure of the program's performance. While accuracy does measure the program's performance, it does not differentiate between malignant tumors that were classified as being benign, and benign tumors that were classified as being malignant. In some applications, the costs associated with all types of errors may be the same. In this problem, however, failing to identify malignant tumors is likely to be a more severe error than mistakenly classifying benign tumors as being malignant.

We can measure each of the possible prediction outcomes to create different views of the classifier's performance. When the system correctly classifies a tumor as being malignant, the prediction is called a **true positive**. When the system incorrectly classifies a benign tumor as being malignant, the prediction is a **false positive**. Similarly, a **false negative** is an incorrect prediction that the tumor is benign, and a **true negative** is a correct prediction that a tumor is benign. These four outcomes can be used to calculate several common measures of classification performance, including **accuracy**, **precision**, and **recall**.

Accuracy is calculated with the following formula, where *TP* is the number of true positives, *TN* is the number of true negatives, *FP* is the number of false positives, and *FN* is the number of false negatives:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision is the fraction of the tumors that were predicted to be malignant that are actually malignant. Precision is calculated with the following formula:

$$P = \frac{TP}{TP + FP}$$

Recall is the fraction of malignant tumors that the system identified. Recall is calculated with the following formula:

$$R = \frac{TP}{TP + FN}$$

In this example, precision measures the fraction of tumors that were predicted to be malignant that are actually malignant. Recall measures the fraction of truly malignant tumors that were detected.

The precision and recall measures could reveal that a classifier with impressive accuracy actually fails to detect most of the malignant tumors. If most tumors are benign, even a classifier that never predicts malignancy could have high accuracy. A different classifier with lower accuracy and higher recall might be better suited to the task, since it will detect more of the malignant tumors.

Many other performance measures for classification can be used; we will discuss some, including metrics for multilabel classification problems, in later chapters. In the next chapter, we will discuss some common performance measures for regression tasks.

# An introduction to scikit-learn

Since its release in 2007, scikit-learn has become one of the most popular open source machine learning libraries for Python. scikit-learn provides algorithms for machine learning tasks including classification, regression, dimensionality reduction, and clustering. It also provides modules for extracting features, processing data, and evaluating models.

Conceived as an extension to the SciPy library, scikit-learn is built on the popular Python libraries NumPy and matplotlib. NumPy extends Python to support efficient operations on large arrays and multidimensional matrices. matplotlib provides visualization tools, and SciPy provides modules for scientific computing.

scikit-learn is popular for academic research because it has a well-documented, easy-to-use, and versatile API. Developers can use scikit-learn to experiment with different algorithms by changing only a few lines of the code. scikit-learn wraps some popular implementations of machine learning algorithms, such as LIBSVM and LIBLINEAR. Other Python libraries, including NLTK, include wrappers for scikit-learn. scikit-learn also includes a variety of datasets, allowing developers to focus on algorithms rather than obtaining and cleaning data.

Licensed under the permissive BSD license, scikit-learn can be used in commercial applications without restrictions. Many of scikit-learn's algorithms are fast and scalable to all but massive datasets. Finally, scikit-learn is noted for its reliability; much of the library is covered by automated tests.

# Installing scikit-learn

This book is written for version 0.15.1 of scikit-learn; use this version to ensure that the examples run correctly. If you have previously installed scikit-learn, you can retrieve the version number with the following code:

```
>>> import sklearn
>>> sklearn.__version__
'0.15.1'
```

If you have not previously installed scikit-learn, you can install it from a package manager or build it from the source. We will review the installation processes for Linux, OS X, and Windows in the following sections, but refer to `http://scikit-learn.org/stable/install.html` for the latest instructions. The following instructions only assume that you have installed Python 2.6, Python 2.7, or Python 3.2 or newer. Go to `http://www.python.org/download/` for instructions on how to install Python.

# Installing scikit-learn on Windows

scikit-learn requires Setuptools, a third-party package that supports packaging and installing software for Python. Setuptools can be installed on Windows by running the bootstrap script at `https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py`.

Windows binaries for the 32- and 64-bit versions of scikit-learn are also available. If you cannot determine which version you need, install the 32-bit version. Both versions depend on NumPy 1.3 or newer. The 32-bit version of NumPy can be downloaded from `http://sourceforge.net/projects/numpy/files/NumPy/`. The 64-bit version can be downloaded from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#scikit-learn`.

A Windows installer for the 32-bit version of scikit-learn can be downloaded from `http://sourceforge.net/projects/scikit-learn/files/`. An installer for the 64-bit version of scikit-learn can be downloaded from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#scikit-learn`.

scikit-learn can also be built from the source code on Windows. Building requires a C/C++ compiler such as MinGW (`http://www.mingw.org/`), NumPy, SciPy, and Setuptools.

To build, clone the Git repository from `https://github.com/scikit-learn/scikit-learn` and execute the following command:

```
python setup.py install
```

# Installing scikit-learn on Linux

There are several options to install scikit-learn on Linux, depending on your distribution. The preferred option to install scikit-learn on Linux is to use `pip`. You may also install it using a package manager, or build scikit-learn from its source.

To install scikit-learn using `pip`, execute the following command:

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

To build scikit-learn, clone the Git repository from `https://github.com/scikit-learn/scikit-learn`. Then install the following dependencies:

```
sudo apt-get install python-dev python-numpy python-numpy-dev python-
setuptools python-numpy-dev python-scipy libatlas-dev g++
```

Navigate to the repository's directory and execute the following command:

```
python setup.py install
```

# Installing scikit-learn on OS X

scikit-learn can be installed on OS X using Macports:

```
sudo port install py26-sklearn
```

If Python 2.7 is installed, run the following command:

```
sudo port install py27-sklearn
```

scikit-learn can also be installed using `pip` with the following command:

```
pip install scikit-learn
```

# Verifying the installation

To verify that scikit-learn has been installed correctly, open a Python console and execute the following:

```
>>> import sklearn
>>> sklearn.__version__
'0.15.1'
```

To run scikit-learn's unit tests, first install the `nose` library. Then execute the following:

```
nosetest sklearn -exe
```

Congratulations! You've successfully installed scikit-learn.

# Installing pandas and matplotlib

pandas is an open source library that provides data structures and analysis tools for Python. pandas is a powerful library, and several books describe how to use pandas for data analysis. We will use a few of panda's convenient tools for importing data and calculating summary statistics.

pandas can be installed on Windows, OS X, and Linux using `pip` with the following command:

```
pip install pandas
```

pandas can also be installed on Debian- and Ubuntu-based Linux distributions using the following command:

```
apt-get install python-pandas
```

matplotlib is a library used to easily create plots, histograms, and other charts with Python. We will use it to visualize training data and models. matplotlib has several dependencies. Like pandas, matplotlib depends on NumPy, which should already be installed. On Debian- and Ubuntu-based Linux distributions, matplotlib and its dependencies can be installed using the following command:

```
apt-get install python-matplotlib
```

Binaries for OS X and Windows can be downloaded from `http://matplotlib.org/downloads.html`.

# Summary

In this chapter we defined machine-learning as the design and study of programs that can improve their performance of a task by learning from experience. We discussed the spectrum of supervision in experience. At one end of the spectrum is supervised learning, in which a program learns from inputs that are labeled with their corresponding outputs. At the opposite end of the spectrum is unsupervised learning, in which the program must discover hidden structure in unlabeled data. Semi-supervised approaches make use of both labeled and unlabeled training data.

We discussed common types of machine learning tasks and reviewed example applications. In classification tasks the program must predict the value of a discrete response variable from the explanatory variables. In regression tasks the program must predict the value of a continuous response variable from the explanatory variables. In regression tasks, the program must predict the value of a continuous response variable from the explanatory variables. Unsupervised learning tasks include clustering, in which observations are organized into groups according to some similarity measure and dimensionality reduction, which reduces a set of explanatory variables to a smaller set of synthetic features that retain as much information as possible. We also reviewed the bias-variance trade-off and discussed common performance measures for different machine learning tasks.

We also discussed the history, goals, and advantages of scikit-learn. Finally, we prepared our development environment by installing scikit-learn and other libraries that are commonly used in conjunction with it. In the next chapter, we will discuss the regression task in more detail, and build our first machine learning model with scikit-learn.

# 2
# Linear Regression

In this chapter you will learn how to use linear models in regression problems. First, we will examine simple linear regression, which models the relationship between a response variable and single explanatory variable. Next, we will discuss multiple linear regression, a generalization of simple linear regression that can support more than one explanatory variable. Then, we will discuss polynomial regression, a special case of multiple linear regression that can effectively model nonlinear relationships. Finally, we will discuss how to train our models by finding the values of their parameters that minimize a cost function. We will work through a toy problem to learn how the models and learning algorithms work before discussing an application with a larger dataset.

## Simple linear regression

In the previous chapter you learned that training data is used to estimate the parameters of a model in supervised learning problems. Past observations of explanatory variables and their corresponding response variables comprise the training data. The model can be used to predict the value of the response variable for values of the explanatory variable that have not been previously observed. Recall that the goal in regression problems is to predict the value of a continuous response variable. In this chapter, we will examine several example linear regression models. We will discuss the training data, model, learning algorithm, and evaluation metrics for each approach. To start, let's consider **simple linear regression**. Simple linear regression can be used to model a linear relationship between one response variable and one explanatory variable. Linear regression has been applied to many important scientific and social problems; the example that we will consider is probably not one of them.

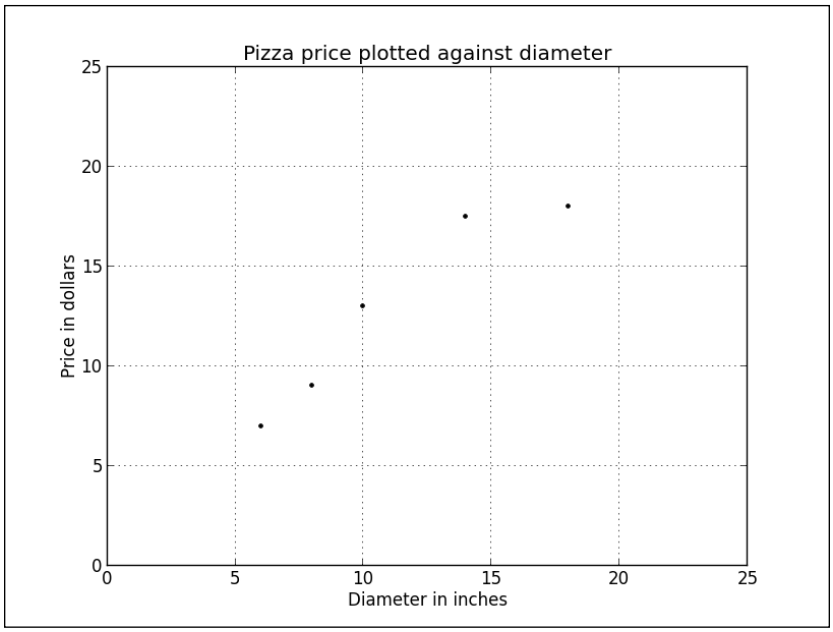
Suppose you wish to know the price of a pizza. You might simply look at a menu. This, however, is a machine learning book, so we will use simple linear regression instead to predict the price of a pizza based on an attribute of the pizza that we can observe. Let's model the relationship between the size of a pizza and its price. First, we will write a program with scikit-learn that can predict the price of a pizza given its size. Then, we will discuss how simple linear regression works and how it can be generalized to work with other types of problems. Let's assume that you have recorded the diameters and prices of pizzas that you have previously eaten in your pizza journal. These observations comprise our training data:

| **Training instance** | **Diameter (in inches)** | **Price (in dollars)** |
|---|---|---|
| 1 | 6 | 7 |
| 2 | 8 | 9 |
| 3 | 10 | 13 |
| 4 | 14 | 17.5 |
| 5 | 18 | 18 |

We can visualize our training data by plotting it on a graph using `matplotlib`:

```
>>> import matplotlib.pyplot as plt
>>> X = [[6], [8], [10], [14],   [18]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> plt.figure()
>>> plt.title('Pizza price plotted against diameter')
>>> plt.xlabel('Diameter in inches')
>>> plt.ylabel('Price in dollars')
>>> plt.plot(X, y, 'k.')
>>> plt.axis([0, 25, 0, 25])
>>> plt.grid(True)
>>> plt.show()
```

The preceding script produces the following graph. The diameters of the pizzas are plotted on the *x* axis and the prices are plotted on the *y* axis.




We can see from the graph of the training data that there is a positive relationship between the diameter of a pizza and its price, which should be corroborated by our own pizza-eating experience. As the diameter of a pizza increases, its price generally increases too. The following pizza-price predictor program models this relationship using linear regression. Let's review the following program and discuss how linear regression works:

```
>>> from sklearn.linear_model import LinearRegression
>>> # Training data
>>> X = [[6], [8], [10], [14],   [18]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> # Create and fit the model
>>> model = LinearRegression()
>>> model.fit(X, y)
>>> print 'A 12" pizza should cost: $%.2f' % model.predict([12])[0]
A 12" pizza should cost: $13.68
```

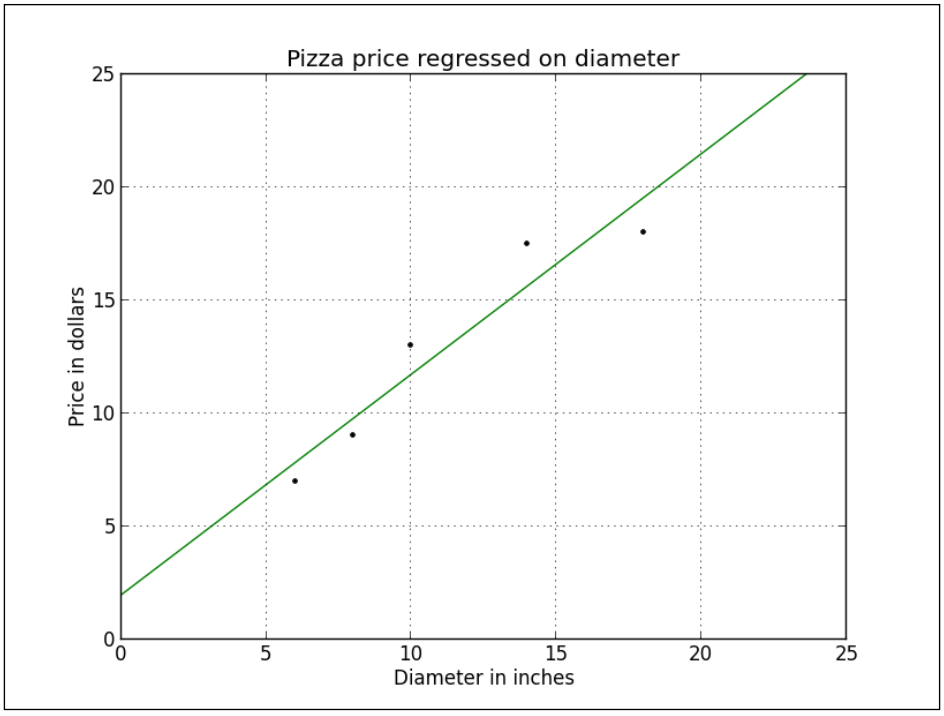
Simple linear regression assumes that a linear relationship exists between the response variable and explanatory variable; it models this relationship with a linear surface called a hyperplane. A hyperplane is a subspace that has one dimension less than the ambient space that contains it. In simple linear regression, there is one dimension for the response variable and another dimension for the explanatory variable, making a total of two dimensions. The regression hyperplane therefore, has one dimension; a hyperplane with one dimension is a line.

The `sklearn.linear_model.LinearRegression` class is an **estimator**. Estimators predict a value based on the observed data. In scikit-learn, all estimators implement the `fit()` and `predict()` methods. The former method is used to learn the parameters of a model, and the latter method is used to predict the value of a response variable for an explanatory variable using the learned parameters. It is easy to experiment with different models using scikit-learn because all estimators implement the `fit` and `predict` methods.

The `fit` method of `LinearRegression` learns the parameters of the following model for simple linear regression:

$$y = \alpha + \beta x$$

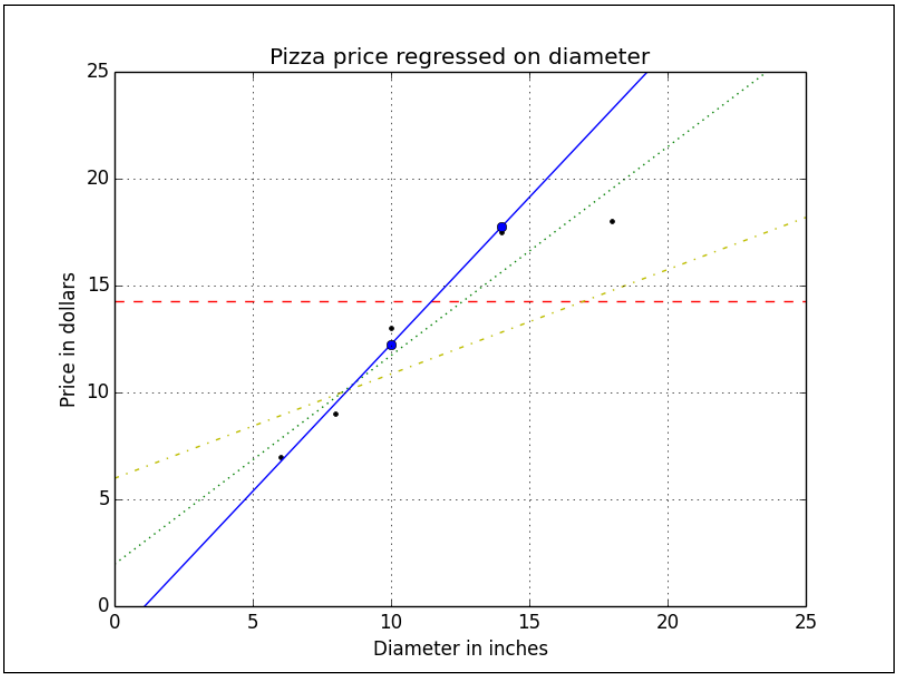
$y$ is the predicted value of the response variable; in this example, it is the predicted price of the pizza. $x$ is the explanatory variable. The intercept term $\alpha$ and coefficient $\beta$ are parameters of the model that are learned by the learning algorithm. The line plotted in the following figure models the relationship between the size of a pizza and its price. Using this model, we would expect the price of an 8-inch pizza to be about $7.33, and the price of a 20-inch pizza to be $18.75.



Using training data to learn the values of the parameters for simple linear regression that produce the best fitting model is called **ordinary least squares** or **linear least squares**. "In this chapter we will discuss methods for approximating the values of the model's parameters and for solving them analytically. First, however, we must define what it means for a model to fit the training data.

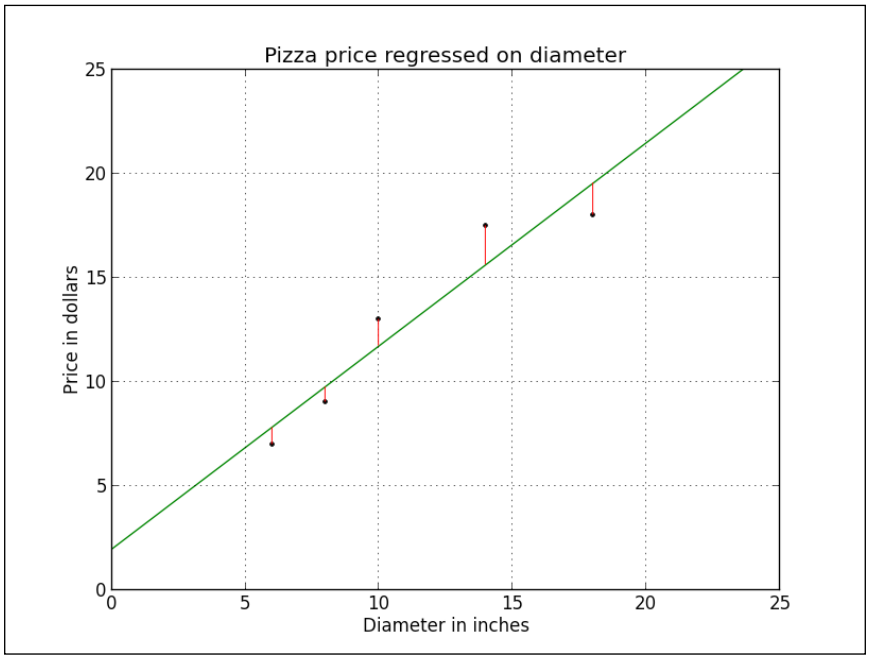
# Evaluating the fitness of a model with a cost function

Regression lines produced by several sets of parameter values are plotted in the following figure. How can we assess which parameters produced the best-fitting regression line?




A **cost function**, also called a **loss function,** is used to define and measure the error of a model. The differences between the prices predicted by the model and the observed prices of the pizzas in the training set are called **residuals** or **training errors**. Later, we will evaluate a model on a separate set of test data; the differences between the predicted and observed values in the test data are called **prediction errors** or **test errors**.

The residuals for our model are indicated by the vertical lines between the points for the training instances and regression hyperplane in the following plot:




We can produce the best pizza-price predictor by minimizing the sum of the residuals. That is, our model fits if the values it predicts for the response variable are close to the observed values for all of the training examples. This measure of the model's fitness is called the **residual sum of squares** cost function. Formally, this function assesses the fitness of a model by summing the squared residuals for all of our training examples. The residual sum of squares is calculated with the formula in the following equation, where $y_i$ is the observed value and $f(x_i)$ is the predicted value:

$$SS_{res} = \sum_{i=1}^{n} \left( y_i - f(x_i) \right)^2$$

Let's compute the residual sum of squares for our model by adding the following two lines to the previous script:

```
>>> import numpy as np
>>> print 'Residual sum of squares: %.2f' % np.mean((model.predict(X)
- y) ** 2)
Residual sum of squares: 1.75
```

Now that we have a cost function, we can find the values of our model's parameters that minimize it.

# Solving ordinary least squares for simple linear regression

In this section, we will work through solving ordinary least squares for simple linear regression. Recall that simple linear regression is given by the following equation:

$$y = \alpha + \beta x$$

Also, recall that our goal is to solve the values of $\beta$ and $\alpha$ that minimize the cost function. We will solve $\beta$ first. To do so, we will calculate the **variance** of $x$ and **covariance** of $x$ and $y$.

Variance is a measure of how far a set of values is spread out. If all of the numbers in the set are equal, the variance of the set is zero. A small variance indicates that the numbers are near the mean of the set, while a set containing numbers that are far from the mean and each other will have a large variance. Variance can be calculated using the following equation:

$$\text{var}(x) = \frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}$$

In the preceding equation, $\overline{x}$ is the mean of $x$, $x_i$ is the value of $x$ for the $ith$ training instance, and $n$ is the number of training instances. Let's calculate the variance of the pizza diameters in our training set:

```
>>> from __future__ import division
>>> xbar = (6 + 8 + 10 + 14 + 18) / 5
>>> variance = ((6 - xbar)**2 + (8 - xbar)**2 + (10 - xbar)**2 + (14 -
xbar)**2 + (18 - xbar)**2) / 4
>>> print variance
23.2
```

NumPy also provides the `var` method to calculate variance. The `ddof` keyword parameter can be used to set Bessel's correction to calculate the sample variance:

```
>>> import numpy as np
>>> print np.var([6, 8, 10, 14, 18], ddof=1)
23.2
```

Covariance is a measure of how much two variables change together. If the value of the variables increase together, their covariance is positive. If one variable tends to increase while the other decreases, their covariance is negative. If there is no linear relationship between the two variables, their covariance will be equal to zero; the variables are linearly uncorrelated but not necessarily independent. Covariance can be calculated using the following formula:

$$\operatorname{cov}(x, y) = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{n-1}$$

As with variance, $x_i$ is the diameter of the $ith$ training instance, $\overline{x}$ is the mean of the diameters, $\overline{y}$ is the mean of the prices, $y_i$ is the price of the $ith$ training instance, and $n$ is the number of training instances. Let's calculate the covariance of the diameters and prices of the pizzas in the training set:

```
>>> xbar = (6 + 8 + 10 + 14 + 18) / 5
>>> ybar = (7 + 9 + 13 + 17.5 + 18) / 5
>>> cov = ((6 - xbar) * (7 - ybar) + (8 - xbar) * (9 - ybar) + (10 -
xbar) * (13 - ybar) +
>>>         (14 - xbar) * (17.5 - ybar) + (18 - xbar) * (18 - ybar)) /
4
>>> print cov
>>> import numpy as np
>>> print np.cov([6, 8, 10, 14, 18], [7, 9, 13, 17.5, 18])[0][1]
22.65
22.65
```

Now that we have calculated the variance of our explanatory variable and the covariance of the response and explanatory variables, we can solve $\beta$ using the following formula:

$$\beta = \frac{\operatorname{cov}(x, y)}{\operatorname{var}(x)}$$

$$\beta = \frac{22.65}{23.2} = 0.9762931034482758$$

Having solved $\beta$, we can solve $\alpha$ using the following formula:

$$\alpha = \overline{y} - \beta\overline{x}$$

In the preceding formula, $\overline{y}$ is the mean of $y$ and $\overline{x}$ is the mean of $x$. $(\overline{x}, \overline{y})$ are the coordinates of the centroid, a point that the model must pass through. We can use the centroid and the value of $\beta$ to solve for $\alpha$ as follows:

$$\alpha = 12.9 - 0.9762931034482758 \times 11.2 = 1.9655172413793114$$

Now that we have solved the values of the model's parameters that minimize the cost function, we can plug in the diameters of the pizzas and predict their prices. For instance, an 11-inch pizza is expected to cost around $12.70, and an 18-inch pizza is expected to cost around $19.54. Congratulations! You used simple linear regression to predict the price of a pizza.

# Evaluating the model

We have used a learning algorithm to estimate a model's parameters from the training data. How can we assess whether our model is a good representation of the real relationship? Let's assume that you have found another page in your pizza journal. We will use the entries on this page as a test set to measure the performance of our model:

| Test Instance | Diameter (in inches) | Observed price (in dollars) | Predicted price (in dollars) |
|---|---|---|---|
| 1 | 8 | 11 | 9.7759 |
| 2 | 9 | 8.5 | 10.7522 |
| 3 | 11 | 15 | 12.7048 |
| 4 | 16 | 18 | 17.5863 |
| 5 | 12 | 11 | 13.6811 |

Several measures can be used to assess our model's predictive capabilities. We will evaluate our pizza-price predictor using **r-squared**. R-squared measures how well the observed values of the response variables are predicted by the model. More concretely, r-squared is the proportion of the variance in the response variable that is explained by the model. An r-squared score of one indicates that the response variable can be predicted without any error using the model. An r-squared score of one half indicates that half of the variance in the response variable can be predicted using the model. There are several methods to calculate r-squared. In the case of simple linear regression, r-squared is equal to the square of the Pearson product moment correlation coefficient, or Pearson's *r*.

Using this method, r-squared must be a positive number between zero and one. This method is intuitive; if r-squared describes the proportion of variance in the response variable explained by the model, it cannot be greater than one or less than zero. Other methods, including the method used by scikit-learn, do not calculate r-squared as the square of Pearson's *r*, and can return a negative r-squared if the model performs extremely poorly. We will follow the method used by scikit-learn to calculate r-squared for our pizza-price predictor.

First, we must measure the total sum of the squares. $y_i$ is the observed value of the response variable for the *ith* test instance, and $\overline{y}$ is the mean of the observed values of the response variable:

$$SS_{tot} = \sum_{i=1}^{n} \left( y_i - \overline{y} \right)^2$$

$$SS_{tot} = \left(11 - 12.7\right)^2 + \left(8.5 - 12.7\right)^2 + \cdots + \left(11 - 12.7\right)^2 = 56.8$$

Next, we must find the residual sum of the squares. Recall that this is also our cost function.

$$SS_{res} = \sum_{i=1}^{n} \left( y_i - f\left(x_i\right) \right)^2$$

$$SS_{res} = \left(11 - 9.7759\right)^2 + \left(8.5 - 10.7522\right)^2 + \cdots + \left(11 - 13.6811\right)^2 = 19.19821359$$

Finally, we can find r-squared using the following formula:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

$$R^2 = 1 - \frac{19.19821359}{56.8} = 0.6620032818661972$$

An r-squared score of **0.6620** indicates that a large proportion of the variance in the test instances' prices is explained by the model. Now, let's confirm our calculation using scikit-learn. The `score` method of `LinearRegression` returns the model's r-squared value, as seen in the following example:

```
>>> from sklearn.linear_model import LinearRegression
>>> X = [[6], [8], [10], [14],   [18]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> X_test = [[8],  [9],   [11], [16], [12]]
>>> y_test = [[11], [8.5], [15], [18], [11]]
>>> model = LinearRegression()
>>> model.fit(X, y)
>>> print 'R-squared: %.4f' % model.score(X_test, y_test)
R-squared: 0.6620
```

# Multiple linear regression

We have trained and evaluated a model to predict the price of a pizza. While you are eager to demonstrate the pizza-price predictor to your friends and co-workers, you are concerned by the model's imperfect r-squared score and the embarrassment its predictions could cause you. How can we improve the model?

Recalling your personal pizza-eating experience, you might have some intuitions about the other attributes of a pizza that are related to its price. For instance, the price often depends on the number of toppings on the pizza. Fortunately, your pizza journal describes toppings in detail; let's add the number of toppings to our training data as a second explanatory variable. We cannot proceed with simple linear regression, but we can use a generalization of simple linear regression that can use multiple explanatory variables called multiple linear regression. Formally, multiple linear regression is the following model:

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

this edit makes no sense. change to "Where simple linear regression uses a single explanatory variable with a single coefficient, multiple linear regression uses a coefficient for each of an arbitrary number of explanatory variables.

$$Y = X\beta$$

For simple linear regression, this is equivalent to the following:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} \alpha + \beta X_1 \\ \alpha + \beta X_2 \\ \vdots \\ \alpha + \beta X_n \end{bmatrix} = \begin{bmatrix} 1 & X_1 \\ 1 & X_2 \\ \vdots & \vdots \\ 1 & X_n \end{bmatrix} \times \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

$Y$ is a column vector of the values of the response variables for the training examples. $\beta$ is a column vector of the values of the model's parameters. $X$, called the design matrix, is an $m \times n$ dimensional matrix of the values of the explanatory variables for the training examples. $m$ is the number of training examples and $n$ is the number of explanatory variables. Let's update our pizza training data to include the number of toppings with the following values:

| Training Example | Diameter (in inches) | Number of toppings | Price (in dollars) |
|---|---|---|---|
| 1 | 6 | 2 | 7 |
| 2 | 8 | 1 | 9 |
| 3 | 10 | 0 | 13 |
| 4 | 14 | 2 | 17.5 |
| 5 | 18 | 0 | 18 |

We must also update our test data to include the second explanatory variable, as follows:

| Test Instance | Diameter (in inches) | Number of toppings | Price (in dollars) |
|---|---|---|---|
| 1 | 8 | 2 | 11 |
| 2 | 9 | 0 | 8.5 |
| 3 | 11 | 2 | 15 |
| 4 | 16 | 2 | 18 |
| 5 | 12 | 0 | 11 |

Our learning algorithm must estimate the values of three parameters: the coefficients for the two features and the intercept term. While one might be tempted to solve $\beta$ by dividing each side of the equation by $X$, division by a matrix is impossible. Just as dividing a number by an integer is equivalent to multiplying by the inverse of the same integer, we can multiply $\beta$ by the inverse of $X$ to avoid matrix division. Matrix inversion is denoted with a superscript -1. Only square matrices can be inverted. $X$ is not likely to be a square; the number of training instances will have to be equal to the number of features for it to be so. We will multiply $X$ by its transpose to yield a square matrix that can be inverted. Denoted with a superscript $T$, the transpose of a matrix is formed by turning the rows of the matrix into columns and vice versa, as follows:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

To recap, our model is given by the following formula:

$$Y = X\beta$$

We know the values of $Y$ and $X$ from our training data. We must find the values of $\beta$, which minimize the cost function. We can solve $\beta$ as follows:

$$\beta = \left(X^T X\right)^{-1} X^T Y$$

We can solve $\beta$ using NumPy, as follows:

```
>>> from numpy.linalg import inv
>>> from numpy import dot, transpose
>>> X = [[1, 6, 2], [1, 8, 1], [1, 10, 0], [1, 14, 2], [1, 18, 0]]
>>> y = [[7], [9], [13], [17.5], [18]]
>>> print dot(inv(dot(transpose(X), X)), dot(transpose(X), y))
[[ 1.1875    ]
 [ 1.01041667]
 [ 0.39583333]]
```

NumPy also provides a least squares function that can solve the values of the parameters more compactly:

```
>>> from numpy.linalg import lstsq
>>> X = [[1, 6, 2], [1, 8, 1], [1, 10, 0], [1, 14, 2], [1, 18, 0]]
>>> y = [[7],    [9],    [13],    [17.5],  [18]]
>>> print lstsq(X, y)[0]
[[ 1.1875    ]
 [ 1.01041667]
 [ 0.39583333]]
```

Let's update our pizza-price predictor program to use the second explanatory variable, and compare its performance on the test set to that of the simple linear regression model:

```
>>> from sklearn.linear_model import LinearRegression
>>> X = [[6, 2], [8, 1], [10, 0], [14, 2], [18, 0]]
>>> y = [[7],    [9],    [13],    [17.5],  [18]]
>>> model = LinearRegression()
>>> model.fit(X, y)
>>> X_test = [[8, 2], [9, 0], [11, 2], [16, 2], [12, 0]]
>>> y_test = [[11],   [8.5],  [15],    [18],    [11]]
>>> predictions = model.predict(X_test)
>>> for i, prediction in enumerate(predictions):
>>>     print 'Predicted: %s, Target: %s' % (prediction, y_test[i])
>>> print 'R-squared: %.2f' % model.score(X_test, y_test)
Predicted: [ 10.0625], Target: [11]
Predicted: [ 10.28125], Target: [8.5]
Predicted: [ 13.09375], Target: [15]
Predicted: [ 18.14583333], Target: [18]
Predicted: [ 13.3125], Target: [11]
R-squared: 0.77
```

It appears that adding the number of toppings as an explanatory variable has improved the performance of our model. In later sections, we will discuss why evaluating a model on a single test set can provide inaccurate estimates of the model's performance, and how we can estimate its performance more accurately by training and testing on many partitions of the data. For now, however, we can accept that the multiple linear regression model performs significantly better than the simple linear regression model. There may be other attributes of pizzas that can be used to explain their prices. What if the relationship between these explanatory variables and the response variable is not linear in the real world? In the next section, we will examine a special case of multiple linear regression that can be used to model nonlinear relationships.

# Polynomial regression

In the previous examples, we assumed that the real relationship between the explanatory variables and the response variable is linear. This assumption is not always true. In this section, we will use **polynomial regression**, a special case of multiple linear regression that adds terms with degrees greater than one to the model. The real-world curvilinear relationship is captured when you transform the training data by adding polynomial terms, which are then fit in the same manner as in multiple linear regression. For ease of visualization, we will again use only one explanatory variable, the pizza's diameter. Let's compare linear regression with polynomial regression using the following datasets:

| Training Instance | Diameter (in inches) | Price (in dollars) |
|---|---|---|
| 1 | 6 | 7 |
| 2 | 8 | 9 |
| 3 | 10 | 13 |
| 4 | 14 | 17.5 |
| 5 | 18 | 18 |

| Testing Instance | Diameter (in inches) | Price (in dollars) |
|---|---|---|
| 1 | 6 | 7 |
| 2 | 8 | 9 |
| 3 | 10 | 13 |
| 4 | 14 | 17.5 |

**Quadratic regression**, or regression with a second order polynomial, is given by the following formula:

$$y = \alpha + \beta_1 x + \beta_2 x^2$$

We are using only one explanatory variable, but the model now has three terms instead of two. The explanatory variable has been transformed and added as a third term to the model to capture the curvilinear relationship. Also, note that the equation for polynomial regression is the same as the equation for multiple linear regression in vector notation. The `PolynomialFeatures` transformer can be used to easily add polynomial features to a feature representation. Let's fit a model to these features, and compare it to the simple linear regression model:

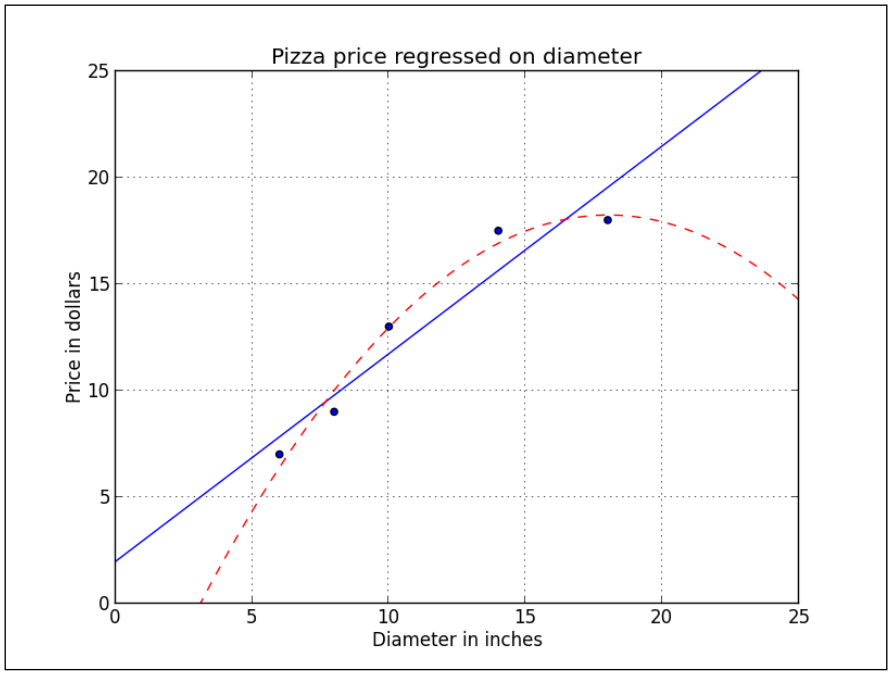
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

```
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.preprocessing import PolynomialFeatures

>>> X_train = [[6], [8], [10], [14],   [18]]
>>> y_train = [[7], [9], [13], [17.5], [18]]
>>> X_test = [[6],  [8],   [11], [16]]
>>> y_test = [[8], [12], [15], [18]]

>>> regressor = LinearRegression()
>>> regressor.fit(X_train, y_train)
>>> xx = np.linspace(0, 26, 100)
>>> yy = regressor.predict(xx.reshape(xx.shape[0], 1))
>>> plt.plot(xx, yy)

>>> quadratic_featurizer = PolynomialFeatures(degree=2)
>>> X_train_quadratic = quadratic_featurizer.fit_transform(X_train)
>>> X_test_quadratic = quadratic_featurizer.transform(X_test)

>>> regressor_quadratic = LinearRegression()
>>> regressor_quadratic.fit(X_train_quadratic, y_train)
>>> xx_quadratic = quadratic_featurizer.transform(xx.reshape(xx.
shape[0], 1))

>>> plt.plot(xx, regressor_quadratic.predict(xx_quadratic), c='r',
linestyle='--')
>>> plt.title('Pizza price regressed on diameter')
>>> plt.xlabel('Diameter in inches')
>>> plt.ylabel('Price in dollars')
>>> plt.axis([0, 25, 0, 25])
>>> plt.grid(True)
>>> plt.scatter(X_train, y_train)
>>> plt.show()

>>> print X_train
>>> print X_train_quadratic
>>> print X_test
>>> print X_test_quadratic
>>> print 'Simple linear regression r-squared', regressor.score(X_
test, y_test)
>>> print 'Quadratic regression r-squared', regressor_quadratic.
score(X_test_quadratic, y_test)
```

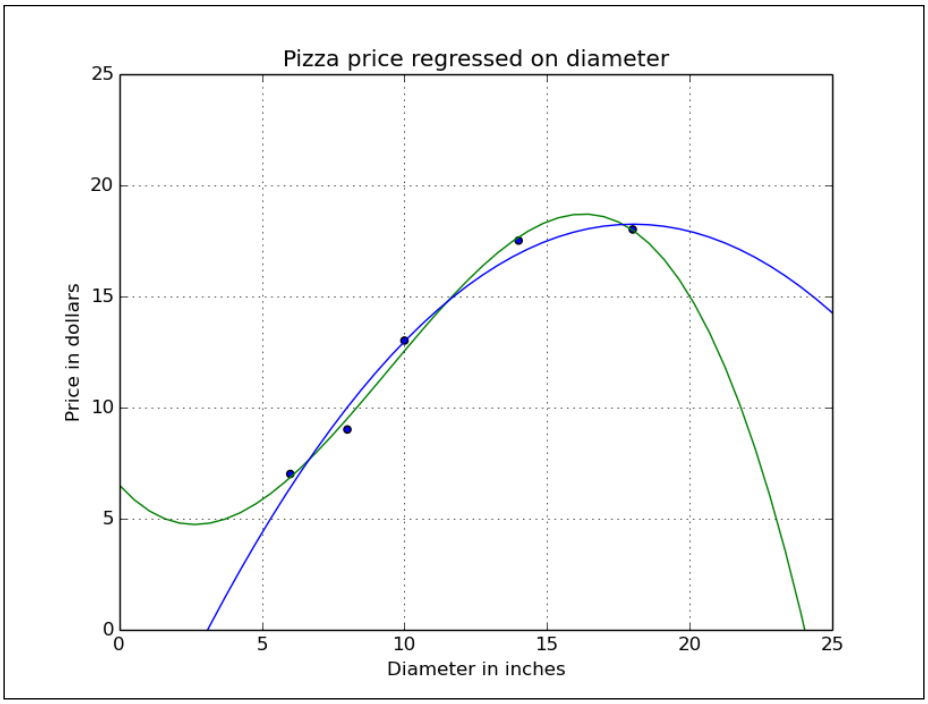
The following is the output of the preceding script:

```
[[6], [8], [10], [14], [18]]
[[  1   6  36]
 [  1   8  64]
 [  1  10 100]
 [  1  14 196]
 [  1  18 324]]
[[6], [8], [11], [16]]
[[  1   6  36]
 [  1   8  64]
 [  1  11 121]
 [  1  16 256]]
Simple linear regression r-squared 0.809726797708
Quadratic regression r-squared 0.867544365635
```

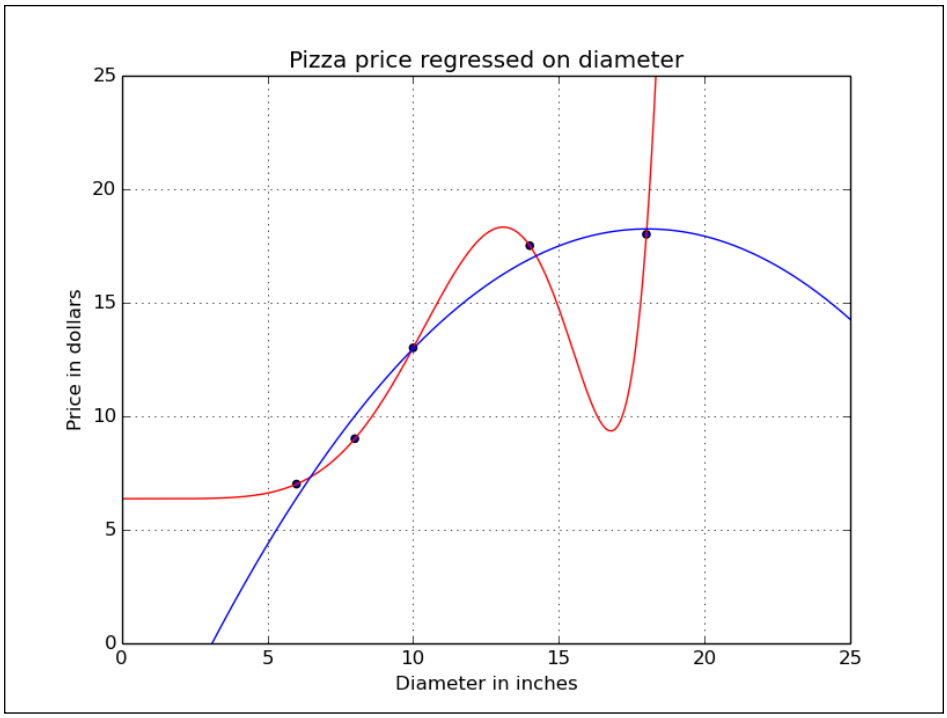
The simple linear regression model is plotted with the solid line in the following figure. Plotted with a dashed line, the quadratic regression model visibly fits the training data better.



The r-squared score of the simple linear regression model is 0.81; the quadratic regression model's r-squared score is an improvement at 0.87. While quadratic and cubic regression models are the most common, we can add polynomials of any degree. The following figure plots the quadratic and cubic models:




Now, let's try an even higher-order polynomial. The plot in the following figure shows a regression curve created by a ninth-degree polynomial:




The ninth-degree polynomial regression model fits the training data almost exactly! The model's r-squared score, however, is -0.09. We created an extremely complex model that fits the training data exactly, but fails to approximate the real relationship. This problem is called **over-fitting**. The model should induce a general rule to map inputs to outputs; instead, it has memorized the inputs and outputs from the training data. As a result, the model performs poorly on test data. It predicts that a 16 inch pizza should cost less than $10, and an 18 inch pizza should cost more than $30. This model exactly fits the training data, but fails to learn the real relationship between size and price.

# Regularization

**Regularization** is a collection of techniques that can be used to prevent over-fitting. Regularization adds information to a problem, often in the form of a penalty against complexity, to a problem. Occam's razor states that a hypothesis with the fewest assumptions is the best. Accordingly, regularization attempts to find the simplest model that explains the data.

scikit-learn provides several regularized linear regression models. **Ridge regression**, also known as **Tikhonov regularization**, penalizes model parameters that become too large. Ridge regression modifies the residual sum of the squares cost function by adding the L2 norm of the coefficients, as follows:

$$RSS_{\text{ridge}} = \sum_{i=1}^{n}\left(y_i - x_i^T\beta\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2$$

$\lambda$ is a hyperparameter that controls the strength of the penalty. **Hyperparameters** are parameters of the model that are not learned automatically and must be set manually. As $\lambda$ increases, the penalty increases, and the value of the cost function increases. When $\lambda$ is equal to zero, ridge regression is equal to linear regression.

scikit-learn also provides an implementation of the **Least Absolute Shrinkage and Selection Operator** (**LASSO**). LASSO penalizes the coefficients by adding their L1 norm to the cost function, as follows:

$$RSS_{\text{lasso}} = \sum_{i=1}^{n}\left(y_i - x_i^T\beta\right)^2 + \lambda\sum_{j=1}^{p}\beta_j$$

The LASSO produces sparse parameters; most of the coefficients will become zero, and the model will depend on a small subset of the features. In contrast, ridge regression produces models in which most parameters are small but nonzero. When explanatory variables are correlated, the LASSO will shrink the coefficients of one variable toward zero. Ridge regression will shrink them more uniformly. Finally, scikit-learn provides an implementation of **elastic net** regularization, which linearly combines the L1 and L2 penalties used by the LASSO and ridge regression. That is, the LASSO and ridge regression are both special cases of the elastic net method in which the hyperparameter for either the L1 or L2 penalty is equal to zero.

# Applying linear regression

We have worked through a toy problem to learn how linear regression models relationships between explanatory and response variables. Now we'll use a real data set and apply linear regression to an important task. Assume that you are at a party, and that you wish to drink the best wine that is available. You could ask your friends for recommendations, but you suspect that they will drink any wine, regardless of its provenance. Fortunately, you have brought pH test strips and other tools to measure various physicochemical properties of wine—it is, after all, a party. We will use machine learning to predict the quality of the wine based on its physicochemical attributes.

The UCI Machine Learning Repository's Wine data set measures eleven physicochemical attributes, including the pH and alcohol content, of 1,599 different red wines. Each wine's quality has been scored by human judges. The scores range from zero to ten; zero is the worst quality and ten is the best quality. The data set can be downloaded from `https://archive.ics.uci.edu/ml/datasets/Wine`. We will approach this problem as a regression task and regress the wine's quality onto one or more physicochemical attributes. The response variable in this problem takes only integer values between 0 and 10; we could view these as discrete values and approach the problem as a multiclass classification task. In this chapter, however, we will view the response variable as a continuous value.

# Exploring the data

| Fixed acidity | Volatile acidity | Citric acidity | Residual sugar | Chlorides | Free sulfur dioxide | Total sulfur dioxide | Density | pH | Sulphates | Alcohol | Quality |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 7.4 | 0.7 | 0 | 1.9 | 0.076 | 11 | 34 | 0.9978 | 3.51 | 0.56 | 9.4 | 5 |
| 7.8 | 0.88 | 0 | 2.6 | 0.098 | 25 | 67 | 0.9968 | 3.2 | 0.68 | 9.8 | 5 |
| 7.8 | 0.76 | 0.04 | 2.3 | 0.092 | 15 | 54 | 0.997 | 3.26 | 0.65 | 9.8 | 5 |
| 11.2 | 0.28 | 0.56 | 1.9 | 0.075 | 17 | 60 | 0.998 | 3.16 | 0.58 | 9.8 | 6 |

scikit-learn is intended to be a tool to build machine learning systems; its capabilities to explore data are impoverished compared to those of packages such as SPSS Statistics or the R language. We will use pandas, an open source data analysis library for Python, to generate descriptive statistics from the data; we will use these statistics to inform some of the design decisions of our model. pandas introduces Python to some concepts from R such as the dataframe, a two-dimensional, tabular, and heterogeneous data structure. Using pandas for data analysis is the topic of several books; we will use only a few basic methods in the following examples.

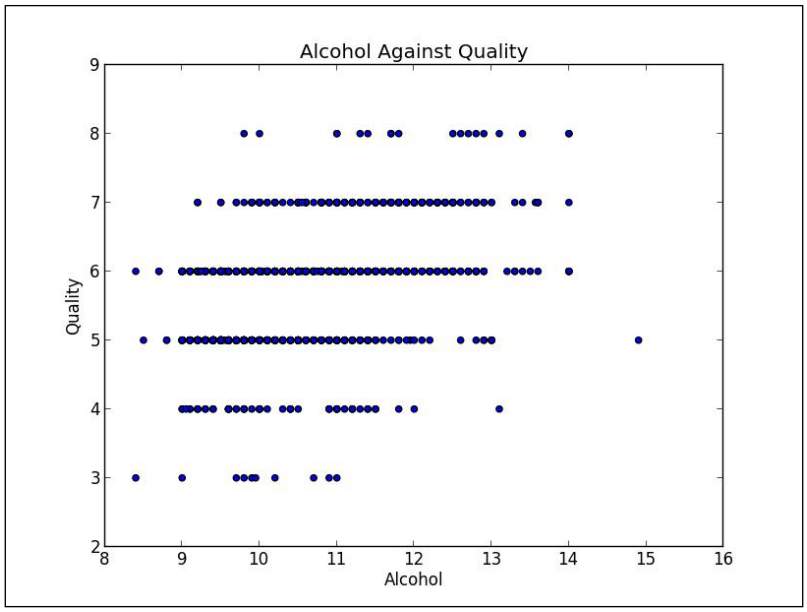
First, we will load the data set and review some basic summary statistics for the variables. The data is provided as a `.csv` file. Note that the fields are separated by semicolons rather than commas):

```
>>> import pandas as pd
>>> df = pd.read_csv('winequality-red.csv', sep=';')
>>> df.describe()

                pH    sulphates      alcohol      quality
count  1599.000000  1599.000000  1599.000000  1599.000000
mean      3.311113     0.658149    10.422983     5.636023
std       0.154386     0.169507     1.065668     0.807569
min       2.740000     0.330000     8.400000     3.000000
25%       3.210000     0.550000     9.500000     5.000000
50%       3.310000     0.620000    10.200000     6.000000
75%       3.400000     0.730000    11.100000     6.000000
max       4.010000     2.000000    14.900000     8.000000
```

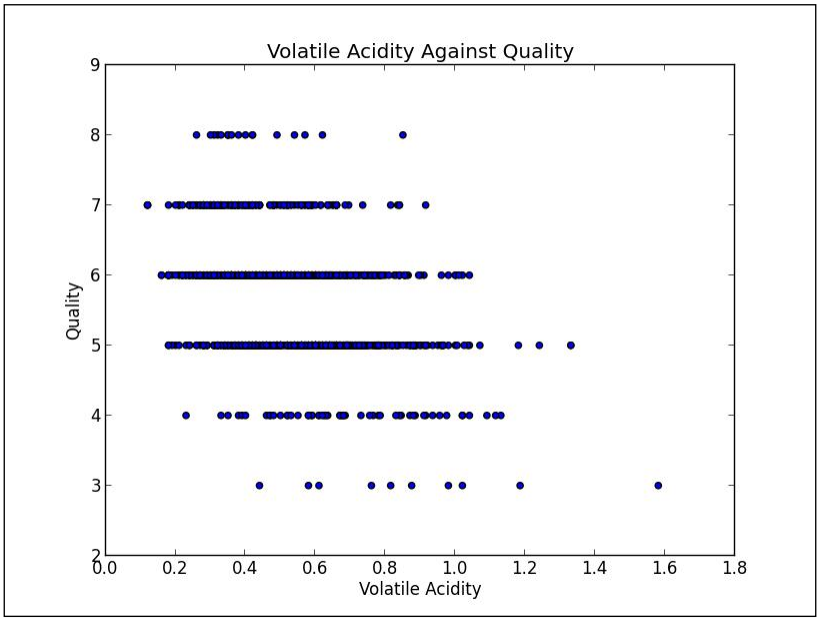
The `pd.read_csv()` function is a convenience utility that loads the `.csv` file into a dataframe. The `Dataframe.describe()` method calculates summary statistics for each column of the dataframe. The preceding code sample shows the summary statistics for only the last four columns of the dataframe. Note the summary for the quality variable; most of the wines scored five or six. Visualizing the data can help indicate if relationships exist between the response variable and the explanatory variables. Let's use `matplotlib` to create some scatter plots. Consider the following code snippet:

```
>>> import matplotlib.pylab as plt
>>> plt.scatter(df['alcohol'], df['quality'])
>>> plt.xlabel('Alcohol')
>>> plt.ylabel('Quality')
>>> plt.title('Alcohol Against Quality')
>>> plt.show()
```

The output of the preceding code snippet is shown in the following figure:




A weak positive relationship between the alcohol content and quality is visible in the scatter plot in the preceding figure; wines that have high alcohol content are often high in quality. The following figure reveals a negative relationship between volatile acidity and quality:



These plots suggest that the response variable depends on multiple explanatory variables; let's model the relationship with multiple linear regression. How can we decide which explanatory variables to include in the model? `Dataframe.corr()` calculates a pairwise correlation matrix. The correlation matrix confirms that the strongest positive correlation is between the alcohol and quality, and that quality is negatively correlated with volatile acidity, an attribute that can cause wine to taste like vinegar. To summarize, we have hypothesized that good wines have high alcohol content and do not taste like vinegar. This hypothesis seems sensible, though it suggests that wine aficionados may have less sophisticated palates than they claim.

# Fitting and evaluating the model

Now we will split the data into training and testing sets, train the regressor, and evaluate its predictions:

```
>>> from sklearn.linear_model import LinearRegression
>>> import pandas as pd
>>> import matplotlib.pylab as plt
>>> from sklearn.cross_validation import train_test_split

>>> df = pd.read_csv('wine/winequality-red.csv', sep=';')
>>> X = df[list(df.columns)[:-1]]
>>> y = df['quality']
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)

>>> regressor = LinearRegression()
>>> regressor.fit(X_train, y_train)
>>> y_predictions = regressor.predict(X_test)
>>> print 'R-squared:', regressor.score(X_test, y_test)
0.345622479617
```

First, we loaded the data using pandas and separated the response variable from the explanatory variables. Next, we used the `train_test_split` function to randomly partition the data into training and test sets. The proportions of the data for both partitions can be specified using keyword arguments. By default, 25 percent of the data is assigned to the test set. Finally, we trained the model and evaluated it on the test set.

The r-squared score of 0.35 indicates that 35 percent of the variance in the test set is explained by the model. The performance might change if a different 75 percent of the data is partitioned to the training set. We can use cross-validation to produce a better estimate of the estimator's performance. Recall from chapter one that each cross-validation round trains and tests different partitions of the data to reduce variability:

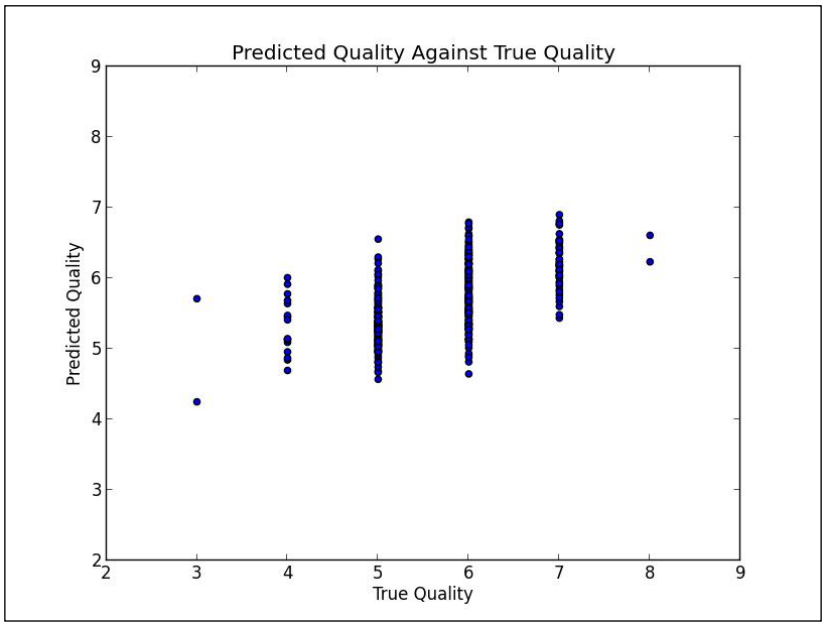
```
>>> import pandas as pd
>>> from sklearn. cross_validation import cross_val_score
>>> from sklearn.linear_model import LinearRegression
>>> df = pd.read_csv('data/winequality-red.csv', sep=';')
>>> X = df[list(df.columns)[:-1]]
>>> y = df['quality']
>>> regressor = LinearRegression()
>>> scores = cross_val_score(regressor, X, y, cv=5)
>>> print scores.mean(), scores
0.290041628842 [ 0.13200871  0.31858135  0.34955348  0.369145
0.2809196 ]
```

The `cross_val_score` helper function allows us to easily perform cross-validation using the provided data and estimator. We specified a five-fold cross validation using the `cv` keyword argument, that is, each instance will be randomly assigned to one of the five partitions. Each partition will be used to train and test the model. `cross_val_score` returns the value of the estimator's `score` method for each round. The r-squared scores range from 0.13 to 0.36! The mean of the scores, 0.29, is a better estimate of the estimator's predictive power than the r-squared score produced from a single train / test split.

Let's inspect some of the model's predictions and plot the true quality scores against the predicted scores:

```
Predicted: 4.89907499467 True: 4
Predicted: 5.60701048317 True: 6
Predicted: 5.92154439575 True: 6
Predicted: 5.54405696963 True: 5
Predicted: 6.07869910663 True: 7
Predicted: 6.036656327 True: 6
Predicted: 6.43923020473 True: 7
Predicted: 5.80270760407 True: 6
Predicted: 5.92425033278 True: 5
Predicted: 5.31809822449 True: 6
Predicted: 6.34837585295 True: 6
```

The following figure shows the output of the preceding code:




As expected, few predictions exactly match the true values of the response variable. The model is also better at predicting the qualities of average wines, since most of the training data is for average wines.

# Fitting models with gradient descent

In the examples in this chapter, we analytically solved the values of the model's parameters that minimize the cost function with the following equation:

$$\beta = \left( X^T X \right)^{-1} X^T Y$$

Recall that $X$ is the matrix of the values of the explanatory variables for each training example. The dot product of $X^T X$ results in a square matrix with dimensions $n \times n$, where $n$ is equal to the number of explanatory variables. The computational complexity of inverting this square matrix is nearly cubic in the number of explanatory variables. While the number of explanatory variables has been small in this chapter's examples, this inversion can be prohibitively costly for problems with tens of thousands of explanatory variables, which we will encounter in the following chapters. Furthermore, $X^T X$ cannot be inverted if its determinant is equal to zero. In this section, we will discuss another method to efficiently estimate the optimal values of the model's parameters called **gradient descent**. Note that our definition of a good fit has not changed; we will still use gradient descent to estimate the values of the model's parameters that minimize the value of the cost function.

Gradient descent is sometimes described by the analogy of a blindfolded man who is trying to find his way from somewhere on a mountainside to the lowest point of the valley. He cannot see the topography, so he takes a step in the direction with the steepest decline. He then takes another step, again in the direction with the steepest decline. The sizes of his steps are proportional to the steepness of the terrain at his current position. He takes big steps when the terrain is steep, as he is confident that he is still near the peak and that he will not overshoot the valley's lowest point. The man takes smaller steps as the terrain becomes less steep. If he were to continue taking large steps, he may accidentally step over the valley's lowest point. He would then need to change direction and step toward the lowest point of the valley again. By taking decreasingly large steps, he can avoid stepping back and forth over the valley's lowest point. The blindfolded man continues to walk until he cannot take a step that will decrease his altitude; at this point, he has found the bottom of the valley.

Formally, gradient descent is an optimization algorithm that can be used to estimate the local minimum of a function. Recall that we are using the residual sum of squares cost function, which is given by the following equation:

$$SS_{res} = \sum_{i=1}^{n} \left( y_i - f\left( x_i \right) \right)^2$$

We can use gradient descent to find the values of the model's parameters that minimize the value of the cost function. Gradient descent iteratively updates the values of the model's parameters by calculating the partial derivative of the cost function at each step. The calculus required to compute the partial derivative of the cost function is beyond the scope of this book, and is also not required to work with scikit-learn. However, having an intuition for how gradient descent works can help you use it effectively.

It is important to note that gradient descent estimates the local minimum of a function. A three-dimensional plot of the values of a convex cost function for all possible values of the parameters looks like a bowl. The bottom of the bowl is the sole local minimum. Non-convex cost functions can have many local minima, that is, the plots of the values of their cost functions can have many peaks and valleys. Gradient descent is only guaranteed to find the local minimum; it will find a valley, but will not necessarily find the lowest valley. Fortunately, the residual sum of the squares cost function is convex.

An important hyperparameter of gradient descent is the learning rate, which controls the size of the blindfolded man's steps. If the learning rate is small enough, the cost function will decrease with each iteration until gradient descent has converged on the optimal parameters. As the learning rate decreases, however, the time required for gradient descent to converge increases; the blindfolded man will take longer to reach the valley if he takes small steps than if he takes large steps. If the learning rate is too large, the man may repeatedly overstep the bottom of the valley, that is, gradient descent could oscillate around the optimal values of the parameters.

There are two varieties of gradient descent that are distinguished by the number of training instances that are used to update the model parameters in each training iteration. **Batch gradient descent**, which is sometimes called only gradient descent, uses all of the training instances to update the model parameters in each iteration. **Stochastic Gradient Descent** (**SGD**), in contrast, updates the parameters using only a single training instance in each iteration. The training instance is usually selected randomly. Stochastic gradient descent is often preferred to optimize cost functions when there are hundreds of thousands of training instances or more, as it will converge more quickly than batch gradient descent. Batch gradient descent is a deterministic algorithm, and will produce the same parameter values given the same training set. As a stochastic algorithm, SGD can produce different parameter estimates each time it is run. SGD may not minimize the cost function as well as gradient descent because it uses only single training instances to update the weights. Its approximation is often close enough, particularly for convex cost functions such as residual sum of squares.

Let's use stochastic gradient descent to estimate the parameters of a model with scikit-learn. `SGDRegressor` is an implementation of SGD that can be used even for regression problems with hundreds of thousands or more features. It can be used to optimize different cost functions to fit different linear models; by default, it will optimize the residual sum of squares. In this example, we will predict the prices of houses in the Boston Housing data set from 13 explanatory variables:

```
>>> import numpy as np
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import SGDRegressor
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.cross_validation import train_test_split
>>> data = load_boston()
>>> X_train, X_test, y_train, y_test = train_test_split(data.data,
data.target)
```

scikit-learn provides a convenience function for loading the data set. First, we split the data into training and testing sets using `train_test_split`:

```
>>> X_scaler = StandardScaler()
>>> y_scaler = StandardScaler()
>>> X_train = X_scaler.fit_transform(X_train)
>>> y_train = y_scaler.fit_transform(y_train)
>>> X_test = X_scaler.transform(X_test)
>>> y_test = y_scaler.transform(y_test)
```

Next, we scaled the features using `StandardScaler`, which we will describe in detail in the next chapter:

```
>>> regressor = SGDRegressor(loss='squared_loss')
>>> scores = cross_val_score(regressor, X_train, y_train, cv=5)
>>> print 'Cross validation r-squared scores:', scores
>>> print 'Average cross validation r-squared score:', np.mean(scores)
>>> regressor.fit_transform(X_train, y_train)
>>> print 'Test set r-squared score', regressor.score(X_test, y_test)
```

Finally, we trained the estimator, and evaluated it using cross validation and the test set. The following is the output of the script:

```
Cross validation r-squared scores: [ 0.73428974  0.80517755
0.58608421  0.83274059  0.69279604]
Average cross validation r-squared score: 0.730217627242
Test set r-squared score 0.653188093125
```

# Summary

In this chapter we discussed three cases of linear regression. We worked through an example of simple linear regression, which models the relationship between a single explanatory variable and a response variable using a line. We then discussed multiple linear regression, which generalizes simple linear regression to model the relationship between multiple explanatory variables and a response variable. Finally, we described polynomial regression, a special case of multiple linear regression that models non-linear relationships between explanatory variables and a response variable. These three models can be viewed as special cases of the generalized linear model, a framework for model linear relationships, which we will discuss in more detail in *Chapter 4*, *From Linear Regression to Logistic Regression*.

We assessed the fitness of models using the residual sum of squares cost function and discussed two methods to learn the values of a model's parameters that minimize the cost function. First, we solved the values of the model's parameters analytically. We then discussed gradient descent, a method that can efficiently estimate the optimal values of the model's parameters even when the model has a large number of features. The features in this chapter's examples were simple measurements of their explanatory variables; it was easy to use them in our models. In the next chapter, you will learn to create features for different types of explanatory variables, including categorical variables, text, and images.

# 3
# Feature Extraction and Preprocessing

The examples discussed in the previous chapter used simple numeric explanatory variables, such as the diameter of a pizza. Many machine learning problems require learning from observations of categorical variables, text, or images. In this chapter, you will learn basic techniques for preprocessing data and creating feature representations of these observations. These techniques can be used with the regression models discussed in *Chapter 2*, *Linear Regression*, as well as the models we will discuss in subsequent chapters.

## Extracting features from categorical variables

Many machine learning problems have **categorical**, or **nominal**, rather than continuous features. For example, an application that predicts a job's salary based on its description might use categorical features such as the job's location. Categorical variables are commonly encoded using **one-of-K** or **one-hot** encoding, in which the explanatory variable is encoded using one binary feature for each of the variable's possible values.

For example, let's assume that our model has a `city` explanatory variable that can take one of three values: `New York`, `San Francisco`, or `Chapel Hill`. One-hot encoding represents this explanatory variable using one binary feature for each of the three possible cities.

In scikit-learn, the `DictVectorizer` class can be used to one-hot encode categorical features:

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> onehot_encoder = DictVectorizer()
>>> instances = [
>>>     {'city': 'New York'},
>>>     {'city': 'San Francisco'},
>>>     {'city': 'Chapel Hill'}>>> ]
>>> print onehot_encoder.fit_transform(instances).toarray()
[[ 0.  1.  0.] [ 0.  0.  1.][ 1.  0.  0.]]
```

Note that resulting features will not necessarily be ordered in the feature vector as they were encountered. In the first training example, the `city` feature's value is `New York`. The second element in the feature vectors corresponds to the `New York` value and is set to `1` for the first instance. It may seem intuitive to represent the values of a categorical explanatory variable with a single integer feature, but this would encode artificial information. For example, the feature vectors for the previous example would have only one dimension. `New York` could be represented by `0`, `San Francisco` by `1`, and `Chapel Hill` by `2`. This representation would encode an order for the values of the variable that does not exist in the real world; there is no natural order of cities.

# Extracting features from text

Many machine learning problems use text as an explanatory variable. Text must be transformed to a different representation that encodes as much of its meaning as possible in a feature vector. In the following sections we will review variations of the most common representation of text that is used in machine learning: the bag-of-words model.

# The bag-of-words representation

The most common representation of text is the **bag-of-words** model. This representation uses a multiset, or bag, that encodes the words that appear in a text; the bag-of-words does not encode any of the text's syntax, ignores the order of words, and disregards all grammar. Bag-of-words can be thought of as an extension to one-hot encoding. It creates one feature for each word of interest in the text. The bag-of-words model is motivated by the intuition that documents containing similar words often have similar meanings. The bag-of-words model can be used effectively for document classification and retrieval despite the limited information that it encodes.

A collection of documents is called a **corpus**. Let's use a corpus with the following two documents to examine the bag-of-words model:

```
corpus = [
    'UNC played Duke in basketball',
    'Duke lost the basketball game'
]
```

This corpus contains eight unique words: `UNC`, `played`, `Duke`, `in`, `basketball`, `lost`, `the`, and `game`. The corpus's unique words comprise its **vocabulary**. The bag-of-words model uses a feature vector with an element for each of the words in the corpus's vocabulary to represent each document. Our corpus has eight unique words, so each document will be represented by a vector with eight elements. The number of elements that comprise a feature vector is called the vector's **dimension**. A **dictionary** maps the vocabulary to indices in the feature vector.

In the most basic bag-of-words representation, each element in the feature vector is a binary value that represents whether or not the corresponding word appeared in the document. For example, the first word in the first document is `UNC`. The first word in the dictionary is `UNC`, so the first element in the vector is equal to one. The last word in the dictionary is `game`. The first document does not contain the word `game`, so the eighth element in its vector is set to `0`. The `CountVectorizer` class can produce a bag-of-words representation from a string or file. By default, `CountVectorizer` converts the characters in the documents to lowercase, and **tokenizes** the documents. Tokenization is the process of splitting a string into **tokens**, or meaningful sequences of characters. Tokens frequently are words, but they may also be shorter sequences including punctuation characters and affixes. The `CountVectorizer` class tokenizes using a regular expression that splits strings on whitespace and extracts sequences of characters that are two or more characters in length.

The documents in our corpus are represented by the following feature vectors:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
>>>     'UNC played Duke in basketball',
>>>     'Duke lost the basketball game'
>>> ]
>>> vectorizer = CountVectorizer()
>>> print vectorizer.fit_transform(corpus).todense()
>>> print vectorizer.vocabulary_
[[1 1 0 1 0 1 0 1]
 [1 1 1 0 1 0 1 0]]
{u'duke': 1, u'basketball': 0, u'lost': 4, u'played': 5, u'game': 2,
u'unc': 7, u'in': 3, u'the': 6}
```

Now let's add a third document to our corpus:

```
corpus = [
    'UNC played Duke in basketball',
    'Duke lost the basketball game',
    'I ate a sandwich'
]
```

Our corpus's dictionary now contains the following ten unique words. Note that `I` and `a` were not extracted as they do not match the default regular expression that CountVectorizer uses to tokenize strings:

```
{u'duke': 2, u'basketball': 1, u'lost': 5, u'played': 6, u'in': 4,
u'game': 3, u'sandwich': 7, u'unc': 9, u'ate': 0, u'the': 8}
```

Now, our feature vectors are as follows:

```
UNC played Duke in basketball = [[0 1 1 0 1 0 1 0 0 1]]
Duke lost the basketball game = [[0 1 1 1 0 1 0 0 1 0]]
I ate a sandwich = [[1 0 0 0 0 0 0 1 0 0]]
```

The meanings of the first two documents are more similar to each other than they are to the third document, and their corresponding feature vectors are more similar to each other than they are to the third document's feature vector when using a metric such as **Euclidean distance**. The Euclidean distance between two vectors is equal to the **Euclidean norm**, or L2 norm, of the difference between the two vectors:

$$d = \left\| x_0 - x_1 \right\|$$

Recall that the Euclidean norm of a vector is equal to the vector's magnitude, which is given by the following equation:

$$\left\| x \right\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$$

scikit-learn's `euclidean_distances` function can be used to calculate the distance between two or more vectors, and it confirms that the most semantically similar documents are also the closest to each other in space. In the following example, we will use the `euclidean_distances` function to compare the feature vectors for our documents:

```
>>> from sklearn.metrics.pairwise import euclidean_distances
>>> counts = [
>>>     [0, 1, 1, 0, 0, 1, 0, 1],
>>>     [0, 1, 1, 1, 1, 0, 0, 0],
```

```
>>>     [1, 0, 0, 0, 0, 0, 1, 0]
>>> ]
>>> print 'Distance between 1st and 2nd documents:', euclidean_
distances(counts[0], counts[1])
>>> print 'Distance between 1st and 3rd documents:', euclidean_
distances(counts[0], counts[2])
>>> print 'Distance between 2nd and 3rd documents:', euclidean_
distances(counts[1], counts[2])
Distance between 1st and 2nd documents: [[ 2.]]
Distance between 1st and 3rd documents: [[ 2.44948974]]
Distance between 2nd and 3rd documents: [[ 2.44948974]]
```

Now let's assume that we are using a corpus of news articles instead of our toy corpus. Our dictionary may now have hundreds of thousands of unique words instead of only twelve. The feature vectors representing the articles will each have hundreds of thousands of elements, and many of the elements will be zero. Most sports articles will not have any of the words particular to finance articles and most culture articles will not have any of the words particular to articles about finance. High-dimensional feature vectors that have many zero-valued elements are called **sparse vectors**.

Using high-dimensional data creates several problems for all machine learning tasks, including those that do not involve text. The first problem is that high-dimensional vectors require more memory than smaller vectors. NumPy provides some data types that mitigate this problem by efficiently representing only the nonzero elements of sparse vectors.

The second problem is known as the **curse of dimensionality**, or the **Hughes effect**. As the feature space's dimensionality increases, more training data is required to ensure that there are enough training instances with each combination of the feature's values. If there are insufficient training instances for a feature, the algorithm may overfit noise in the training data and fail to generalize. In the following sections, we will review several strategies to reduce the dimensionality of text features. In *Chapter 7, Dimensionality Reduction with PCA*, we will review techniques for numerical dimensionality reduction.

# Stop-word filtering

A basic strategy to reduce the dimensions of the feature space is to convert all of the text to lowercase. This is motivated by the insight that the letter case does not contribute to the meanings of most words; `sandwich` and `Sandwich` have the same meaning in most contexts. Capitalization may indicate that a word is at the beginning of a sentence, but the bag-of-words model has already discarded all information from word order and grammar.

A second strategy is to remove words that are common to most of the documents in the corpus. These words, called **stop words**, include determiners such as `the`, `a`, and `an`; auxiliary verbs such as `do`, `be`, and `will`; and prepositions such as `on`, `around`, and `beneath`. Stop words are often functional words that contribute to the document's meaning through grammar rather than their denotations. The `CountVectorizer` class can filter stop words provided as the `stop_words` keyword argument and also includes a basic English stop list. Let's recreate the feature vectors for our documents using stop filtering:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
>>>     'UNC played Duke in basketball',
>>>     'Duke lost the basketball game',
>>>     'I ate a sandwich'
>>> ]
>>> vectorizer = CountVectorizer(stop_words='english')
>>> print vectorizer.fit_transform(corpus).todense()
>>> print vectorizer.vocabulary_
[[0 1 1 0 0 1 0 1]
 [0 1 1 1 1 0 0 0]
 [1 0 0 0 0 0 1 0]]
{u'duke': 2, u'basketball': 1, u'lost': 4, u'played': 5, u'game': 3,
u'sandwich': 6, u'unc': 7, u'ate': 0}
```

The feature vectors have now fewer dimensions, and the first two document vectors are still more similar to each other than they are to the third document.

# Stemming and lemmatization

While stop filtering is an easy strategy for dimensionality reduction, most stop lists contain only a few hundred words. A large corpus may still have hundreds of thousands of unique words after filtering. Two similar strategies for further reducing dimensionality are called **stemming** and **lemmatization**.

A high-dimensional document vector may separately encode several derived or inflected forms of the same word. For example, `jumping` and `jumps` are both forms of the word `jump`; a document vector in a corpus of long-jumping articles may encode each inflected form with a separate element in the feature vector. Stemming and lemmatization are two strategies to condense inflected and derived forms of a word into a single feature.

Let's consider another toy corpus with two documents:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
>>>     'He ate the sandwiches',
>>>     'Every sandwich was eaten by him'
>>> ]
>>> vectorizer = CountVectorizer(binary=True, stop_words='english')
>>> print vectorizer.fit_transform(corpus).todense()
>>> print vectorizer.vocabulary_
[[1 0 0 1]
 [0 1 1 0]]
{u'sandwich': 2, u'ate': 0, u'sandwiches': 3, u'eaten': 1}
```

The documents have similar meanings, but their feature vectors have no elements in common. Both documents contain a conjugation of `ate` and an inflected form of `sandwich`. Ideally, these similarities should be reflected in the feature vectors. Lemmatization is the process of determining the **lemma**, or the morphological root, of an inflected word based on its context. Lemmas are the base forms of words that are used to key the word in a dictionary. **Stemming** has a similar goal to lemmatization, but it does not attempt to produce the morphological roots of words. Instead, stemming removes all patterns of characters that appear to be affixes, resulting in a token that is not necessarily a valid word. Lemmatization frequently requires a lexical resource, like WordNet, and the word's part of speech. Stemming algorithms frequently use rules instead of lexical resources to produce stems and can operate on any token, even without its context.

Let's consider lemmatization of the word `gathering` in two documents:

```
corpus = [
    'I am gathering ingredients for the sandwich.',
    'There were many wizards at the gathering.'
]
```

In the first sentence `gathering` is a verb, and its lemma is `gather`. In the second sentence `gathering` is a noun, and its lemma is `gathering`. We will use the **Natural Language Tool Kit** (**NTLK**) to stem and lemmatize the corpus. NLTK can be installed using the instructions at `http://www.nltk.org/install.html`. After installation, execute the following code:

```
>>> import nltk
>>> nltk.download()
```

Then follow the instructions to download the corpora for NLTK.

Using the parts of speech of `gathering`, NLTK's `WordNetLemmatizer` correctly lemmatizes the words in both documents as shown in the following example:

```
>>> from nltk.stem.wordnet import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> print lemmatizer.lemmatize('gathering', 'v')
>>> print lemmatizer.lemmatize('gathering', 'n')
gather
gathering
```

Let's compare lemmatization with stemming. The Porter stemmer cannot consider the inflected form's part of speech and returns `gather` for both documents:

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()
>>> print stemmer.stem('gathering')
gather
```

Now let's lemmatize our toy corpus:

```
>>> from nltk import word_tokenize
>>> from nltk.stem import PorterStemmer
>>> from nltk.stem.wordnet import WordNetLemmatizer
>>> from nltk import pos_tag
>>> wordnet_tags = ['n', 'v']
>>> corpus = [
>>>     'He ate the sandwiches',
>>>     'Every sandwich was eaten by him'
>>> ]
>>> stemmer = PorterStemmer()
>>> print 'Stemmed:', [[stemmer.stem(token) for token in word_
tokenize(document)] for document in corpus]
>>> def lemmatize(token, tag):
>>>     if tag[0].lower() in ['n', 'v']:
>>>         return lemmatizer.lemmatize(token, tag[0].lower())
>>>     return token
>>> lemmatizer = WordNetLemmatizer()
>>> tagged_corpus = [pos_tag(word_tokenize(document)) for document in
corpus]
>>> print 'Lemmatized:', [[lemmatize(token, tag) for token, tag in
document] for document in tagged_corpus]
Stemmed: [['He', 'ate', 'the', 'sandwich'], ['Everi', 'sandwich',
'wa', 'eaten', 'by', 'him']]
Lemmatized: [['He', 'eat', 'the', 'sandwich'], ['Every', 'sandwich',
'be', 'eat', 'by', 'him']]
```

Through stemming and lemmatization, we reduced the dimensionality of our feature space. We produced feature representations that more effectively encode the meanings of the documents despite the fact that the words in the corpus's dictionary are inflected differently in the sentences.

# Extending bag-of-words with TF-IDF weights

In the previous section we used the bag-of-words representation to create feature vectors that encode whether or not a word from the corpus's dictionary appears in a document. These feature vectors do not encode grammar, word order, or the frequencies of words. It is intuitive that the frequency with which a word appears in a document could indicate the extent to which a document pertains to that word. A long document that contains one occurrence of a word may discuss an entirely different topic than a document that contains many occurrences of the same word. In this section, we will create feature vectors that encode the frequencies of words, and discuss strategies to mitigate two problems caused by encoding term frequencies.

Instead of using a binary value for each element in the feature vector, we will now use an integer that represents the number of times that the words appeared in the document.

We will use the following corpus. With stop word filtering, the corpus is represented by the following feature vector:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = ['The dog ate a sandwich, the wizard transfigured a
sandwich, and I ate a sandwich']
>>> vectorizer = CountVectorizer(stop_words='english')
>>> print vectorizer.fit_transform(corpus).todense()
[[2 1 3 1 1]]
{u'sandwich': 2, u'wizard': 4, u'dog': 1, u'transfigured': 3, u'ate':
0}
```

The element for `dog` is now set to `1` and the element for `sandwich` is set to `2` to indicate that the corresponding words occurred once and twice, respectively. Note that the `binary` keyword argument of `CountVectorizer` is omitted; its default value is `False`, which causes it to return raw term frequencies rather than binary frequencies. Encoding the terms' raw frequencies in the feature vector provides additional information about the meanings of the documents, but assumes that all of the documents are of similar lengths.

Many words might appear with the same frequency in two documents, but the documents could still be dissimilar if one document is many times larger than the other. scikit-learn's `TfdfTransformer` object can mitigate this problem by transforming a matrix of term frequency vectors into a matrix of normalized term frequency weights. By default, `TfdfTransformer` smoothes the raw counts and applies L2 normalization. The smoothed, normalized term frequencies are given by the following equation:

$$tf(t,d) = \frac{f(t,d)+1}{\|x\|}$$

$f(t,d)$ is the frequency of term $t$ in document $d$ and $\|x\|$ is the L2 norm of the count vector. In addition to normalizing raw term counts, we can improve our feature vectors by calculating **logarithmically scaled term frequencies**, which scale the counts to a more limited range, or **augmented term frequencies**, which further mitigates the bias for longer documents. Logarithmically scaled term frequencies are given by the following equation:

$$tf(t,d) = \log\left(f(t,d)+1\right)$$

The `TfdfTransformer` object calculates logarithmically scaled term frequencies when its `sublinear_tf` keyword argument is set to `True`. Augmented frequencies are given by the following equation:

$$tf(t,d) = 0.5 + \frac{0.5 * f(t,d)}{\max f(w,d) : w \in d}$$

$\max\{f(w,d) : w \in d\}$ is the greatest frequency of all of the words in document $d$. scikit-learn 0.15.2 does not implement augmented term frequencies, but the output of `CountVectorizer` can be easily transformed.

Normalization, logarithmically scaled term frequencies, and augmented term frequencies can represent the frequencies of terms in a document while mitigating the effects of different document sizes. However, another problem remains with these representations. The feature vectors contain large weights for terms that occur frequently in a document, even if those terms occur frequently in most documents in the corpus. These terms do not help to represent the meaning of a particular document relative to the rest of the corpus. For example, most of the documents in a corpus of articles about Duke's basketball team could include the words `basketball`, `Coach K`, and `flop`. These words can be thought of as corpus-specific stop words and may not be useful to calculate the similarity of documents. The **inverse document frequency** (**IDF**) is a measure of how rare or common a word is in a corpus. The inverse document frequency is given by the following equation:

$$idf(t, D) = \log \frac{N}{1 + |\, d \in D : t \in d \,|}$$

Here, $N$ is the total number of documents in the corpus and $d \in D : t \in d$ is the number of documents in the corpus that contain the term $t$. A term's **TF-IDF** value is the product of its term frequency and inverse document frequency. `TfidfTransformer` returns TF-IDF's weight when its `use_idf` keyword argument is set to its default value, `True`. Since TF-IDF weighted feature vectors are commonly used to represent text, scikit-learn provides a `TfidfVectorizer` class that wraps `CountVectorizer` and `TfidfTransformer`. Let's use `TfidfVectorizer` to create TF-IDF weighted feature vectors for our corpus:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = [
>>>     'The dog ate a sandwich and I ate a sandwich',
>>>     'The wizard transfigured a sandwich'
>>> ]
>>> vectorizer = TfidfVectorizer(stop_words='english')
>>> print vectorizer.fit_transform(corpus).todense()
[[ 0.75458397  0.37729199  0.53689271  0.          0.        ]
 [ 0.          0.          0.44943642  0.6316672   0.6316672 ]]
```

By comparing the TF-IDF weights to the raw term frequencies, we can see that words that are common to many of the documents in the corpus, such as `sandwich`, have been penalized.

# Space-efficient feature vectorizing with the hashing trick

In this chapter's previous examples, a dictionary containing all of the corpus's unique tokens is used to map a document's tokens to the elements of a feature vector. Creating this dictionary has two drawbacks. First, two passes are required over the corpus: the first pass is used to create the dictionary and the second pass is used to create feature vectors for the documents. Second, the dictionary must be stored in memory, which could be prohibitive for large corpora. It is possible to avoid creating this dictionary through applying a hash function to the token to determine its index in the feature vector directly. This shortcut is called the **hashing trick**. The following example uses `HashingVectorizer` to demonstrate the hashing trick:

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> corpus = ['the', 'ate', 'bacon', 'cat']
>>> vectorizer = HashingVectorizer(n_features=6)
>>> print vectorizer.transform(corpus).todense()
[[-1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0.  0.]]
```

The hashing trick is stateless. It can be used to create feature vectors in both parallel and online, or streaming, applications because it does not require an initial pass over the corpus.. Note that `n_features` is an optional keyword argument. Its default value, $2^{20}$, is adequate for most problems; it is set to `6` here so that the matrix will be small enough to print and still display all of the nonzero features. Also, note that some of the term frequencies are negative. Since hash collisions are possible, `HashingVectorizer` uses a signed hash function. The value of a feature takes the same sign as its token's hash; if the term `cats` appears twice in a document and is hashed to `-3`, the fourth element of the document's feature vector will be decremented by two. If the term `dogs` also appears twice and is hashed to `3`, the fourth element of the feature vector will be incremented by two. Using a signed hash function creates the possibility that errors from hash collisions will cancel each other out rather than accumulate; a loss of information is preferable to a loss of information and the addition of spurious information. Another disadvantage of the hashing trick is that the resulting model is more difficult to inspect, as the hashing function cannot recall what input token is mapped to each element of the feature vector.

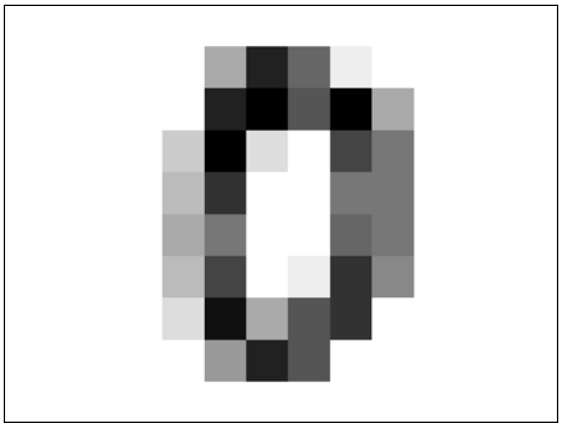
# Extracting features from images

Computer vision is the study and design of computational artifacts that process and understand images. These artifacts sometimes employ machine learning. An overview of computer vision is far beyond the scope of this book, but in this section we will review some basic techniques used in computer vision to represent images in machine learning problems.

# Extracting features from pixel intensities

A digital image is usually a raster, or pixmap, that maps colors to coordinates on a grid. An image can be viewed as a matrix in which each element represents a color. A basic feature representation for an image can be constructed by reshaping the matrix into a vector by concatenating its rows together. **Optical character recognition** (**OCR**) is a canonical machine learning problem. Let's use this technique to create basic feature representations that could be used in an OCR application for recognizing hand-written digits in character-delimited forms.

The `digits` dataset included with scikit-learn contains grayscale images of more than 1,700 hand-written digits between zero and nine. Each image has eight pixels on a side. Each pixel is represented by an intensity value between zero and 16; white is the most intense and is indicated by zero, and black is the least intense and is indicated by 16. The following figure is an image of a hand-written digit taken from the dataset:



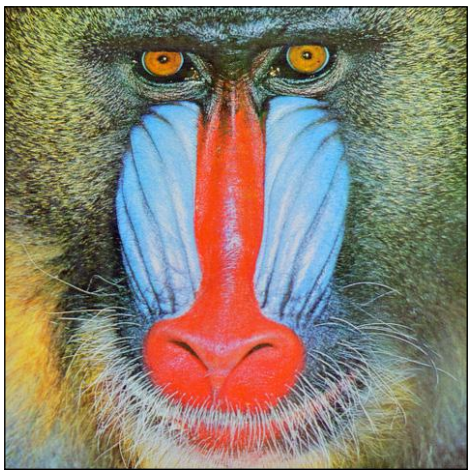
Let's create a feature vector for the image by reshaping its 8 x 8 matrix into a 64-dimensional vector:

```
>>> from sklearn import datasets
>>> digits = datasets.load_digits()
>>> print 'Digit:', digits.target[0]
>>> print digits.images[0]
>>> print 'Feature vector:\n', digits.images[0].reshape(-1, 64)
Digit: 0
[[  0.   0.   5.  13.   9.   1.   0.   0.]
 [  0.   0.  13.  15.  10.  15.   5.   0.]
 [  0.   3.  15.   2.   0.  11.   8.   0.]
 [  0.   4.  12.   0.   0.   8.   8.   0.]
 [  0.   5.   8.   0.   0.   9.   8.   0.]
 [  0.   4.  11.   0.   1.  12.   7.   0.]
 [  0.   2.  14.   5.  10.  12.   0.   0.]
 [  0.   0.   6.  13.  10.   0.   0.   0.]]
Feature vector:
[[  0.   0.   5.  13.   9.   1.   0.   0.   0.   0.  13.  15.  10.
15.
    5.   0.   0.   3.  15.   2.   0.  11.   8.   0.   0.   4.  12.
0.
    0.   8.   8.   0.   0.   5.   8.   0.   0.   9.   8.   0.   0.
4.
   11.   0.   1.  12.   7.   0.   0.   2.  14.   5.  10.  12.   0.
0.
    0.   0.   6.  13.  10.   0.   0.   0.]]
```

This representation can be effective for some basic tasks, like recognizing printed characters. However, recording the intensity of every pixel in the image produces prohibitively large feature vectors. A tiny 100 x 100 grayscale image would require a 10,000-dimensional vector, and a 1920 x 1080 grayscale image would require a 2,073,600-dimensional vector. Unlike the TF-IDF feature vectors we created, in most problems these vectors are not sparse. Space-complexity is not the only disadvantage of this representation; learning from the intensities of pixels at particular locations results in models that are sensitive to changes in the scale, rotation, and translation of images. A model trained on our basic feature representations might not be able to recognize the same zero if it were shifted a few pixels in any direction, enlarged, or rotated a few degrees. Furthermore, learning from pixel intensities is itself problematic, as the model can become sensitive to changes in illumination. For these reasons, this representation is ineffective for tasks that involve photographs or other natural images. Modern computer vision applications frequently use either hand-engineered feature extraction methods that are applicable to many different problems, or automatically learn features without supervision problem using techniques such as deep learning. We will focus on the former in the next section.

# Extracting points of interest as features

The feature vector we created previously represents every pixel in the image; all of the informative attributes of the image are represented and all of the noisy attributes are represented too. After inspecting the training data, we can see that all of the images have a perimeter of white pixels; these pixels are not useful features. Humans can quickly recognize many objects without observing every attribute of the object. We can recognize a car from the contours of the hood without observing the rear-view mirrors, and we can recognize an image of a human face from a nose or mouth. This intuition is motivation to create representations of only the most informative attributes of an image. These informative attributes, or **points of interest**, are points that are surrounded by rich textures and can be reproduced despite perturbing the image. **Edges** and **corners** are two common types of points of interest. An edge is a boundary at which pixel intensity rapidly changes, and a corner is an intersection of two edges. Let's use scikit-image to extract points of interest from the following figure:

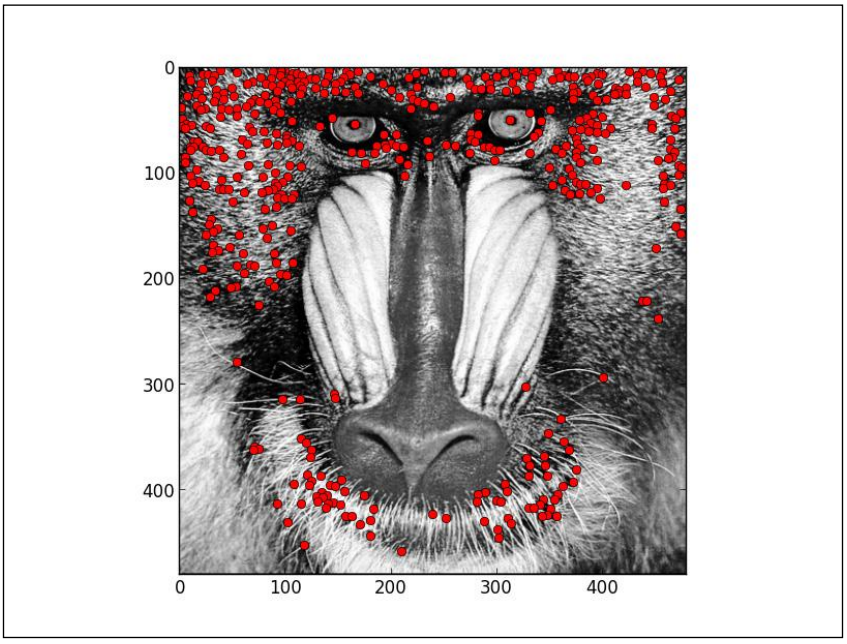


The code to extract the points of interest is as follows:

```
>>> import numpy as nps
>>> from skimage.feature import corner_harris, corner_peaks
>>> from skimage.color import rgb2gray
>>> import matplotlib.pyplot as plt
>>> import skimage.io as io
>>> from skimage.exposure import equalize_hist

>>> def show_corners(corners, image):
>>>     fig = plt.figure()
>>>     plt.gray()
>>>     plt.imshow(image)
```

```
>>>     y_corner, x_corner = zip(*corners)
>>>     plt.plot(x_corner, y_corner, 'or')
>>>     plt.xlim(0, image.shape[1])
>>>     plt.ylim(image.shape[0], 0)
>>>     fig.set_size_inches(np.array(fig.get_size_inches()) * 1.5)
>>>     plt.show()

>>> mandrill = io.imread('/home/gavin/PycharmProjects/mastering-
machine-learning/ch4/img/mandrill.png')
>>> mandrill = equalize_hist(rgb2gray(mandrill))
>>> corners = corner_peaks(corner_harris(mandrill), min_distance=2)
>>> show_corners(corners, mandrill)
```

The following figure plots the extracted points of interest. Of the image's 230400 pixels, 466 were extracted as points of interest. This representation is much more compact; ideally, there is enough variation proximal to the points of interest to reproduce them despite changes in the image's illumination.



# SIFT and SURF

**Scale-Invariant Feature Transform** (**SIFT**) is a method for extracting features from an image that is less sensitive to the scale, rotation, and illumination of the image than the extraction methods we have previously discussed. Each SIFT feature, or descriptor, is a vector that describes edges and corners in a region of an image. Unlike the points of interest in our previous example, SIFT also captures information about the composition of each point of interest and its surroundings. **Speeded-Up Robust Features** (**SURF**) is another method of extracting interesting points of an image and creating descriptions that are invariant of the image's scale, orientation, and illumination. SURF can be computed more quickly than SIFT, and it is more effective at recognizing features across images that have been transformed in certain ways.

Explaining how SIFT and SURF extraction are implemented is beyond the scope of this book. However, with an intuition for how they work, we can still effectively use libraries that implement them.

In this example, we will extract SURF from the following image using the `mahotas` library.

Like the extracted points of interest, the extracted SURF are only the first step in creating a feature representation that could be used in a machine learning task. Different SURF will be extracted for each instance in the training set. In *Chapter 6*, *Clustering with K-Means*, we will cluster extracted SURF to learn features that can be used by an image classifier. In the following example we will use `mahotas` to extract SURF descriptors:

```
>>> import mahotas as mh
>>> from mahotas.features import surf

>>> image = mh.imread('zipper.jpg', as_grey=True)
>>> print 'The first SURF descriptor:\n', surf.surf(image)[0]
>>> print 'Extracted %s SURF descriptors' % len(surf.surf(image))
The first SURF descriptor:
[  6.73839947e+02   2.24033945e+03   3.18074483e+00   2.76324459e+03
  -1.00000000e+00   1.61191475e+00   4.44035121e-05   3.28041690e-04
   2.44845817e-04   3.86297608e-04  -1.16723672e-03  -8.81290243e-04
   1.65414959e-03   1.28393061e-03  -7.45077384e-04   7.77655540e-04
   1.16078772e-03   1.81434398e-03   1.81736394e-04  -3.13096961e-04
    3.06559785e-04   3.43443699e-04   2.66200498e-04  -5.79522387e-04
   1.17893036e-03   1.99547411e-03  -2.25938217e-01  -1.85563853e-01
   2.27973631e-01   1.91510135e-01  -2.49315698e-01   1.95451021e-01
   2.59719480e-01   1.98613061e-01  -7.82458546e-04   1.40287015e-03
   2.86712113e-03   3.15971628e-03   4.98444730e-04  -6.93986983e-04
   1.87531652e-03   2.19041521e-03   1.80681053e-01  -2.70528820e-01
   2.32414943e-01   2.72932870e-01   2.65725332e-01   3.28050743e-01
   2.98609869e-01   3.41623138e-01   1.58078002e-03  -4.67968721e-04
   2.35704122e-03   2.26279888e-03   6.43115065e-06   1.22501486e-04
   1.20064616e-04   1.76564805e-04   2.14148537e-03   8.36243899e-05
   2.93382280e-03   3.10877776e-03   4.53469215e-03  -3.15254535e-04
   6.92437341e-03   3.56880279e-03  -1.95228401e-04   3.73674995e-05
   7.02700555e-04   5.45156362e-04]
Extracted 994 SURF descriptors
```

# Data standardization

Many estimators perform better when they are trained on standardized data sets. Standardized data has **zero mean** and **unit variance**. An explanatory variable with zero mean is centered about the origin; its average value is zero. A feature vector has unit variance when the variances of its features are all of the same order of magnitude. For example, assume that a feature vector encodes two explanatory variables. The first values of the first variable range from zero to one. The values of the second explanatory variable range from zero to 100,000. The second feature must be scaled to a range closer to {0,1} for the data to have unit variance. If a feature's variance is orders of magnitude greater than the variances of the other features, that feature may dominate the learning algorithm and prevent it from learning from the other variables. Some learning algorithms also converge to the optimal parameter values more slowly when data is not standardized. The value of an explanatory variable can be standardized by subtracting the variable's mean and dividing the difference by the variable's standard deviation. Data can be easily standardized using scikit-learn's `scale` function:

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X = np.array([
>>>     [0., 0., 5., 13., 9., 1.],
>>>     [0., 0., 13., 15., 10., 15.],
>>>     [0., 3., 15., 2., 0., 11.]
>>> ])
>>> print preprocessing.scale(X)
[[ 0.         -0.70710678 -1.38873015  0.52489066  0.59299945
-1.35873244]
 [ 0.         -0.70710678  0.46291005  0.87481777  0.81537425
1.01904933]
 [ 0.          1.41421356  0.9258201  -1.39970842 -1.4083737
0.33968311]]
```

# Summary

In this chapter, we discussed feature extraction and developed an understanding about the basic techniques for transforming arbitrary data into feature representations that can be used by machine learning algorithms. First, we created features from categorical explanatory variables using one-hot encoding and scikit-learn's `DictVectorizer`. Then, we discussed the creation of feature vectors for one of the most common types of data used in machine learning problems: text. We worked through several variations of the bag-of-words model, which discards all syntax and encodes only the frequencies of the tokens in a document. We began by creating basic binary term frequencies with `CountVectorizer`. You learned to preprocess text by filtering stop words and stemming tokens, and you also replaced the term counts in our feature vectors with TF-IDF weights that penalize common words and normalize for documents of different lengths. Next, we created feature vectors for images. We began with an optical character recognition problem in which we represented images of hand-written digits with flattened matrices of pixel intensities. This is a computationally costly approach. We improved our representations of images by extracting only their most interesting points as SURF descriptors.

Finally, you learned to standardize data to ensure that our estimators can learn from all of the explanatory variables and can converge as quickly as possible. We will use these feature extraction techniques in the subsequent chapters' examples. In the next chapter, we will combine the bag-of-words representation with a generalization of multiple linear regressions to classify documents.

# 4
# From Linear Regression to Logistic Regression

In *Chapter 2*, *Linear Regression*, we discussed simple linear regression, multiple linear regression, and polynomial regression. These models are special cases of the generalized linear model, a flexible framework that requires fewer assumptions than ordinary linear regression. In this chapter, we will discuss some of these assumptions as they relate to another special case of the generalized linear model called **logistic regression**.

Unlike the models we discussed previously, logistic regression is used for classification tasks. Recall that the goal in classification tasks is to find a function that maps an observation to its associated class or label. A learning algorithm must use pairs of feature vectors and their corresponding labels to induce the values of the mapping function's parameters that produce the best classifier, as measured by a particular performance metric. In binary classification, the classifier must assign instances to one of the two classes. Examples of binary classification include predicting whether or not a patient has a particular disease, whether or not an audio sample contains human speech, or whether or not the Duke men's basketball team will lose in the first round of the NCAA tournament. In multiclass classification, the classifier must assign one of several labels to each instance. In multilabel classification, the classifier must assign a subset of the labels to each instance. In this chapter, we will work through several classification problems using logistic regression, discuss performance measures for the classification task, and apply some of the feature extraction techniques you learned in the previous chapter.

# Binary classification with logistic regression

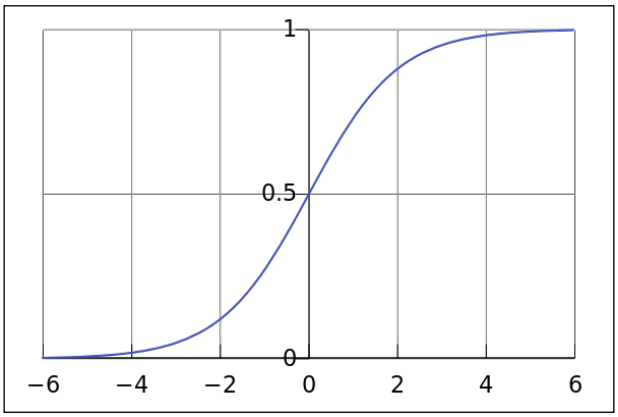
Ordinary linear regression assumes that the response variable is normally distributed. The **normal distribution**, also known as the **Gaussian distribution** or **bell curve**, is a function that describes the probability that an observation will have a value between any two real numbers. Normally distributed data is symmetrical. That is, half of the values are greater than the mean and the other half of the values are less than the mean. The mean, median, and mode of normally distributed data are also equal. Many natural phenomena approximately follow normal distributions. For instance, the height of people is normally distributed; most people are of average height, a few are tall, and a few are short.

In some problems the response variable is not normally distributed. For instance, a coin toss can result in two outcomes: heads or tails. The **Bernoulli distribution** describes the probability distribution of a random variable that can take the positive case with probability *P* or the negative case with probability *1-P*. If the response variable represents a probability, it must be constrained to the range {0,1}. Linear regression assumes that a constant change in the value of an explanatory variable results in a constant change in the value of the response variable, an assumption that does not hold if the value of the response variable represents a probability. Generalized linear models remove this assumption by relating a linear combination of the explanatory variables to the response variable using a link function. In fact, we already used a link function in *Chapter 2*, *Linear Regression*; ordinary linear regression is a special case of the generalized linear model that relates a linear combination of the explanatory variables to a normally distributed response variable using the **identity link function**. We can use a different link function to relate a linear combination of the explanatory variables to the response variable that is not normally distributed.

In logistic regression, the response variable describes the probability that the outcome is the positive case. If the response variable is equal to or exceeds a discrimination threshold, the positive class is predicted; otherwise, the negative class is predicted. The response variable is modeled as a function of a linear combination of the explanatory variables using the **logistic function**. Given by the following equation, the logistic function always returns a value between zero and one:

$$F(t) = \frac{1}{1 + e^{-t}}$$

The following is a plot of the value of the logistic function for the range {-6,6}:




For logistic regression, $t$ is equal to a linear combination of explanatory variables, as follows:

$$F(t) = \frac{1}{1 + e^{-(\beta_0 + \beta_x)}}$$

The **logit function** is the inverse of the logistic function. It links $F(x)$ back to a linear combination of the explanatory variables:

$$g(x) = ln \frac{F(x)}{1 - F(x)} = \beta_0 + \beta_x$$

Now that we have defined the model for logistic regression, let's apply it to a binary classification task.

# Spam filtering

Our first problem is a modern version of the canonical binary classification problem: spam classification. In our version, however, we will classify spam and ham SMS messages rather than e-mail. We will extract TF-IDF features from the messages using techniques you learned in *Chapter 3*, *Feature Extraction and Preprocessing*, and classify the messages using logistic regression.

We will use the SMS Spam Classification Data Set from the UCI Machine Learning Repository. The dataset can be downloaded from `http://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection`. First, let's explore the data set and calculate some basic summary statistics using pandas:

```
>>> import pandas as pd
>>> df = pd.read_csv('data/SMSSpamCollection', delimiter='\t',
header=None)
>>> print df.head()

      0                                                   1
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
[5 rows x 2 columns]

>>> print 'Number of spam messages:', df[df[0] == 'spam'][0].count()
>>> print 'Number of ham messages:', df[df[0] == 'ham'][0].count()

Number of spam messages: 747
Number of ham messages: 4825
```

A binary label and a text message comprise each row. The data set contains 5,574 instances; 4,827 messages are ham and the remaining 747 messages are spam. The ham messages are labeled with zero, and the spam messages are labeled with one. While the noteworthy, or case, outcome is often assigned the label one and the non-case outcome is often assigned zero, these assignments are arbitrary. Inspecting the data may reveal other attributes that should be captured in the model. The following selection of messages characterizes both of the classes:

```
Spam: Free entry in 2 a wkly comp to win FA Cup final tkts 21st May
2005. Text FA to 87121 to receive entry question(std txt rate)T&C's
apply 08452810075over18's
Spam: WINNER!! As a valued network customer you have been selected
to receivea £900 prize reward! To claim call 09061701461. Claim code
KL341. Valid 12 hours only.
Ham: Sorry my roommates took forever, it ok if I come by now?
Ham: Finished class where are you.
```

Let's make some predictions using scikit-learn's `LogisticRegression` class:

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from sklearn.linear_model.logistic import LogisticRegression
>>> from sklearn.cross_validation import train_test_split, cross_val_
score
```

First, we load the `.csv` file using pandas and split the data set into training and test sets. By default, `train_test_split()` assigns 75 percent of the samples to the training set and allocates the remaining 25 percent of the samples to the test set:

```
>>> df = pd.read_csv('data/SMSSpamCollection', delimiter='\t',
header=None)
>>> X_train_raw, X_test_raw, y_train, y_test = train_test_split(df[1],
df[0])
```

Next, we create a `TfidfVectorizer`. Recall from *Chapter 3*, *Feature Extraction and Preprocessing*, that `TfidfVectorizer` combines `CountVectorizer` and `TfidfTransformer`. We fit it with the training messages, and transform both the training and test messages:

```
>>> vectorizer = TfidfVectorizer()
>>> X_train = vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
```

Finally, we create an instance of `LogisticRegression` and train our model. Like `LinearRegression`, `LogisticRegression` implements the `fit()` and `predict()` methods. As a sanity check, we printed a few predictions for manual inspection:

```
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> predictions = classifier.predict(X_test)
>>> for i, prediction in enumerate(predictions[:5]):
>>>     print 'Prediction: %s. Message: %s' % (prediction, X_test_
raw[i])
```

The following is the output of the script:

```
Prediction: ham. Message: If you don't respond imma assume you're
still asleep and imma start calling n shit
Prediction: spam. Message: HOT LIVE FANTASIES call now 08707500020
Just 20p per min NTT Ltd, PO Box 1327 Croydon CR9 5WB 0870 is a
national rate call
```

```
Prediction: ham. Message: Yup... I havent been there before... You
want to go for the yoga? I can call up to book
Prediction: ham. Message: Hi, can i please get a  &lt;#&gt;  dollar
loan from you. I.ll pay you back by mid february. Pls.
Prediction: ham. Message: Where do you need to go to get it?
```

How well does our classifier perform? The performance metrics we used for linear regression are inappropriate for this task. We are only interested in whether the predicted class was correct, not how far it was from the decision boundary. In the next section, we will discuss some performance metrics that can be used to evaluate binary classifiers.

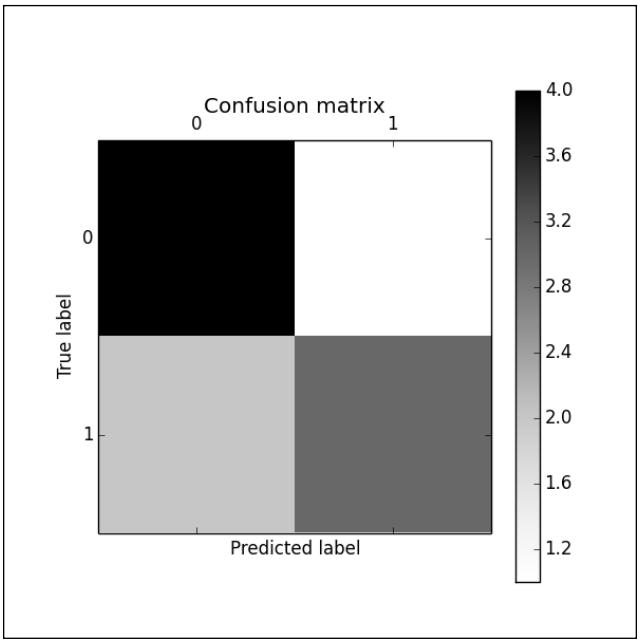
# Binary classification performance metrics

A variety of metrics exist to evaluate the performance of binary classifiers against trusted labels. The most common metrics are **accuracy**, **precision**, **recall**, **F1 measure**, and **ROC AUC score**. All of these measures depend on the concepts of **true positives**, **true negatives**, **false positives**, and **false negatives**. *Positive* and *negative* refer to the classes. *True* and *false* denote whether the predicted class is the same as the true class.

For our SMS spam classifier, a true positive prediction is when the classifier correctly predicts that a message is spam. A true negative prediction is when the classifier correctly predicts that a message is ham. A prediction that a ham message is spam is a false positive prediction, and a spam message incorrectly classified as ham is a false negative prediction. A **confusion matrix**, or **contingency table**, can be used to visualize true and false positives and negatives. The rows of the matrix are the true classes of the instances, and the columns are the predicted classes of the instances:

```
>>> from sklearn.metrics import confusion_matrix
>>> import matplotlib.pyplot as plt

>>> y_test = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
>>> y_pred = [0, 1, 0, 0, 0, 0, 0, 1, 1, 1]
>>> confusion_matrix = confusion_matrix(y_test, y_pred)
>>> print(confusion_matrix)
>>> plt.matshow(confusion_matrix)
>>> plt.title('Confusion matrix')
>>> plt.colorbar()
>>> plt.ylabel('True label')
>>> plt.xlabel('Predicted label')
>>> plt.show()

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

The confusion matrix indicates that there were four true negative predictions, three true positive predictions, two false negative predictions, and one false positive prediction. Confusion matrices become more useful in multi-class problems, in which it can be difficult to determine the most frequent types of errors.




# Accuracy

Accuracy measures a fraction of the classifier's predictions that are correct. scikit-learn provides a function to calculate the accuracy of a set of predictions given the correct labels:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred, y_true = [0, 1, 1, 0], [1, 1, 1, 1]
>>> print 'Accuracy:', accuracy_score(y_true, y_pred)

Accuracy: 0.5
```

`LogisticRegression.score()` predicts and scores labels for a test set using accuracy. Let's evaluate our classifier's accuracy:

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from sklearn.linear_model.logistic import LogisticRegression
>>> from sklearn.cross_validation import train_test_split, cross_val_
score
>>> df = pd.read_csv('data/sms.csv')
>>> X_train_raw, X_test_raw, y_train, y_test = train_test_
split(df['message'], df['label'])
>>> vectorizer = TfidfVectorizer()
>>> X_train = vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> scores = cross_val_score(classifier, X_train, y_train, cv=5)
>>> print np.mean(scores), scores

Accuracy 0.956217208018 [ 0.96057348  0.95334928  0.96411483
0.95454545  0.94850299]
```

Note that your accuracy may differ as the training and test sets are assigned randomly. While accuracy measures the overall correctness of the classifier, it does not distinguish between false positive errors and false negative errors. Some applications may be more sensitive to false negatives than false positives, or vice versa. Furthermore, accuracy is not an informative metric if the proportions of the classes are skewed in the population. For example, a classifier that predicts whether or not credit card transactions are fraudulent may be more sensitive to false negatives than to false positives. To promote customer satisfaction, the credit card company may prefer to risk verifying legitimate transactions than risk ignoring a fraudulent transaction. Because most transactions are legitimate, accuracy is not an appropriate metric for this problem. A classifier that always predicts that transactions are legitimate could have a high accuracy score, but would not be useful. For these reasons, classifiers are often evaluated using two additional measures called precision and recall.

# Precision and recall

Recall from *Chapter 1*, *The Fundamentals of Machine Learning*, that precision is the fraction of positive predictions that are correct. For instance, in our SMS spam classifier, precision is the fraction of messages classified as spam that are actually spam. Precision is given by the following ratio:

$$P = \frac{TP}{TP + FP}$$

Sometimes called sensitivity in medical domains, recall is the fraction of the truly positive instances that the classifier recognizes. A recall score of one indicates that the classifier did not make any false negative predictions. For our SMS spam classifier, recall is the fraction of spam messages that were truly classified as spam. Recall is calculated with the following ratio:

$$R = \frac{TP}{TP + FN}$$

Individually, precision and recall are seldom informative; they are both incomplete views of a classifier's performance. Both precision and recall can fail to distinguish classifiers that perform well from certain types of classifiers that perform poorly. A trivial classifier could easily achieve a perfect recall score by predicting positive for every instance. For example, assume that a test set contains ten positive examples and ten negative examples. A classifier that predicts positive for every example will achieve a recall of one, as follows:

$$R = \frac{10}{10 + 0} = 1$$

A classifier that predicts negative for every example, or that makes only false positive and true negative predictions, will achieve a recall score of zero. Similarly, a classifier that predicts that only a single instance is positive and happens to be correct will achieve perfect precision.

scikit-learn provides a function to calculate the precision and recall for a classifier from a set of predictions and the corresponding set of trusted labels. Let's calculate our SMS classifier's precision and recall:

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from sklearn.linear_model.logistic import LogisticRegression
>>> from sklearn.cross_validation import train_test_split, cross_val_
score
>>> df = pd.read_csv('data/sms.csv')
>>> X_train_raw, X_test_raw, y_train, y_test = train_test_
split(df['message'], df['label'])
>>> vectorizer = TfidfVectorizer()
>>> X_train = vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> precisions = cross_val_score(classifier, X_train, y_train, cv=5,
scoring='precision')
>>> print 'Precision', np.mean(precisions), precisions
>>> recalls = cross_val_score(classifier, X_train, y_train, cv=5,
scoring='recall')
>>> print 'Recalls', np.mean(recalls), recalls

Precision 0.992137651822 [ 0.98717949  0.98666667  1.
0.98684211  1.        ]
Recall 0.677114261885 [ 0.7         0.67272727  0.6         0.68807339
0.72477064]
```

Our classifier's precision is 0.992; almost all of the messages that it predicted as spam were actually spam. Its recall is lower, indicating that it incorrectly classified approximately 22 percent of the spam messages as ham. Your precision and recall may vary since the training and test data are randomly partitioned.

# Calculating the F1 measure

The F1 measure is the harmonic mean, or weighted average, of the precision and recall scores. Also called the f-measure or the f-score, the F1 score is calculated using the following formula:

$$F1 = 2\frac{PR}{P+R}$$

The F1 measure penalizes classifiers with imbalanced precision and recall scores, like the trivial classifier that always predicts the positive class. A model with perfect precision and recall scores will achieve an F1 score of one. A model with a perfect precision score and a recall score of zero will achieve an F1 score of zero. As for precision and recall, scikit-learn provides a function to calculate the F1 score for a set of predictions. Let's compute our classifier's F1 score. The following snippet continues the previous code sample:

```
>>> f1s = cross_val_score(classifier, X_train, y_train, cv=5,
scoring='f1')
>>> print 'F1', np.mean(f1s), f1s

F1 0.80261302628 [ 0.82539683  0.8         0.77348066  0.83157895
0.7826087 ]
```

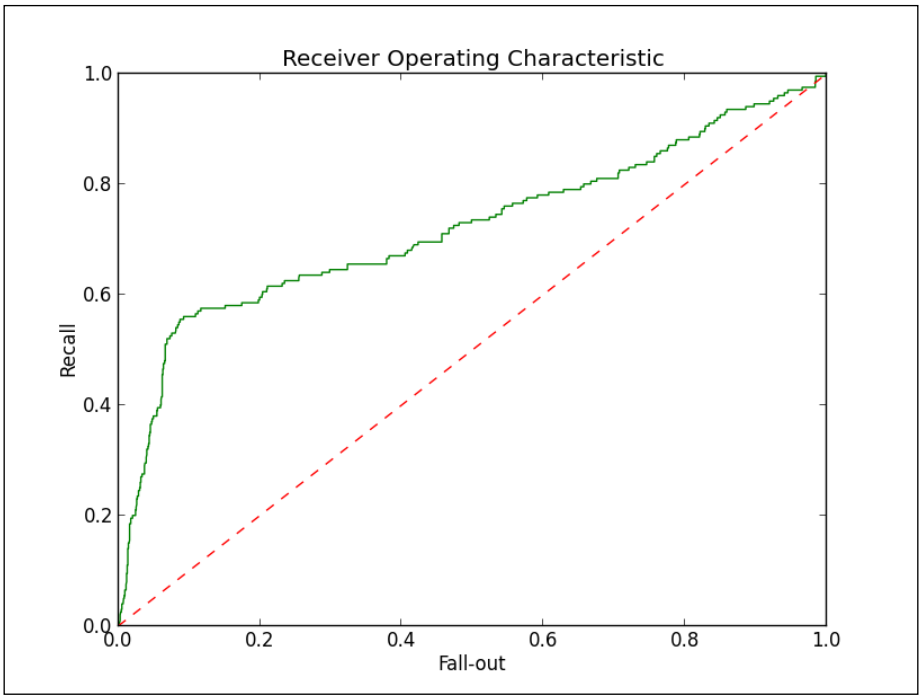
The arithmetic mean of our classifier's precision and recall scores is 0.803. As the difference between the classifier's precision and recall is small, the F1 measure's penalty is small. Models are sometimes evaluated using the F0.5 and F2 scores, which favor precision over recall and recall over precision, respectively.

# ROC AUC

A **Receiver Operating Characteristic**, or **ROC curve**, visualizes a classifier's performance. Unlike accuracy, the ROC curve is insensitive to data sets with unbalanced class proportions; unlike precision and recall, the ROC curve illustrates the classifier's performance for all values of the discrimination threshold. ROC curves plot the classifier's recall against its **fall-out**. Fall-out, or the false positive rate, is the number of false positives divided by the total number of negatives. It is calculated using the following formula:

$$F = \frac{FP}{TN + FP}$$

**AUC** is the area under the ROC curve; it reduces the ROC curve to a single value, which represents the expected performance of the classifier. The dashed line in the following figure is for a classifier that predicts classes randomly; it has an AUC of 0.5. The solid curve is for a classifier that outperforms random guessing:

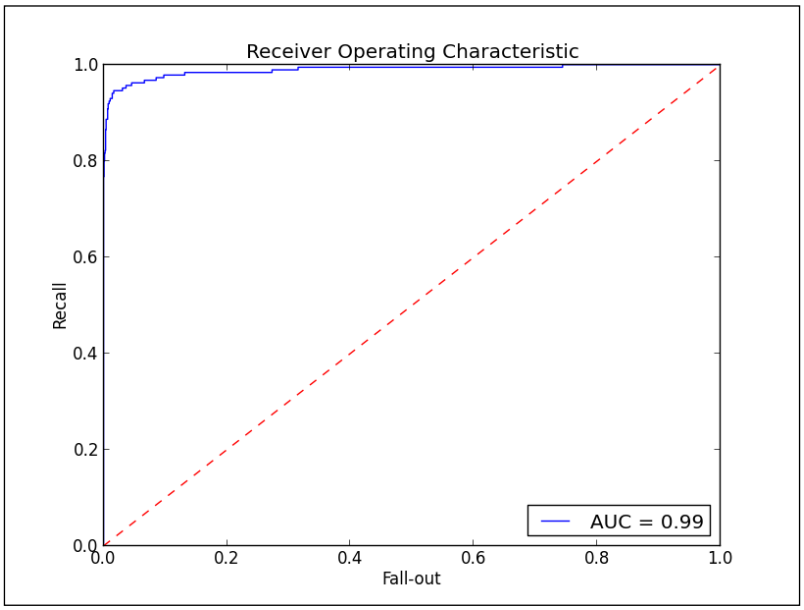



Let's plot the ROC curve for our SMS spam classifier:

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from sklearn.linear_model.logistic import LogisticRegression
>>> from sklearn.cross_validation import train_test_split, cross_val_
score
>>> from sklearn.metrics import roc_curve, auc
>>> df = pd.read_csv('data/sms.csv')
>>> X_train_raw, X_test_raw, y_train, y_test = train_test_
split(df['message'], df['label'])
>>> vectorizer = TfidfVectorizer()
```

```
>>> X_train = vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> predictions = classifier.predict_proba(X_test)
>>> false_positive_rate, recall, thresholds = roc_curve(y_test,
predictions[:, 1])
>>> roc_auc = auc(false_positive_rate, recall)
>>> plt.title('Receiver Operating Characteristic')
>>> plt.plot(false_positive_rate, recall, 'b', label='AUC = %0.2f' %
roc_auc)
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1], 'r--')
>>> plt.xlim([0.0, 1.0])
>>> plt.ylim([0.0, 1.0])
>>> plt.ylabel('Recall')
>>> plt.xlabel('Fall-out')
>>> plt.show()
```

From the ROC AUC plot, it is apparent that our classifier outperforms random guessing; most of the plot area lies under its curve:



# Tuning models with grid search

Hyperparameters are parameters of the model that are not learned. For example, hyperparameters of our logistic regression SMS classifier include the value of the regularization term and thresholds used to remove words that appear too frequently or infrequently. In scikit-learn, hyperparameters are set through the model's constructor. In the previous examples, we did not set any arguments for `LogisticRegression()`; we used the default values for all of the hyperparameters. These default values are often a good start, but they may not produce the optimal model. **Grid search** is a common method to select the hyperparameter values that produce the best model. Grid search takes a set of possible values for each hyperparameter that should be tuned, and evaluates a model trained on each element of the Cartesian product of the sets. That is, grid search is an exhaustive search that trains and evaluates a model for each possible combination of the hyperparameter values supplied by the developer. A disadvantage of grid search is that it is computationally costly for even small sets of hyperparameter values. Fortunately, it is an **embarrassingly parallel** problem; many models can easily be trained and evaluated concurrently since no synchronization is required between the processes. Let's use scikit-learn's `GridSearchCV()` function to find better hyperparameter values:

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model.logistic import LogisticRegression
from sklearn.grid_search import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.cross_validation import train_test_split
from sklearn.metrics import precision_score, recall_score, accuracy_
score

pipeline = Pipeline([
    ('vect', TfidfVectorizer(stop_words='english')),
    ('clf', LogisticRegression())
])
parameters = {
    'vect__max_df': (0.25, 0.5, 0.75),
    'vect__stop_words': ('english', None),
    'vect__max_features': (2500, 5000, 10000, None),
    'vect__ngram_range': ((1, 1), (1, 2)),
    'vect__use_idf': (True, False),
    'vect__norm': ('l1', 'l2'),
    'clf__penalty': ('l1', 'l2'),
    'clf__C': (0.01, 0.1, 1, 10),
}
```

`GridSearchCV()` takes an estimator, a parameter space, and performance measure. The argument `n_jobs` specifies the maximum number of concurrent jobs; set `n_jobs` to `-1` to use all CPU cores. Note that `fit()` must be called in a Python `main` block in order to fork additional processes; this example must be executed as a script, and not in an interactive interpreter:

```
if __name__ == "__main__":
    grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1,
verbose=1, scoring='accuracy', cv=3)
    df = pd.read_csv('data/sms.csv')
    X, y, = df['message'], df['label']
    X_train, X_test, y_train, y_test = train_test_split(X, y)
    grid_search.fit(X_train, y_train)
    print 'Best score: %0.3f' % grid_search.best_score_
    print 'Best parameters set:'
    best_parameters = grid_search.best_estimator_.get_params()
    for param_name in sorted(parameters.keys()):
        print '\t%s: %r' % (param_name, best_parameters[param_name])
    predictions = grid_search.predict(X_test)
    print 'Accuracy:', accuracy_score(y_test, predictions)
    print 'Precision:', precision_score(y_test, predictions)
    print 'Recall:', recall_score(y_test, predictions)
```

The following is the output of the script:

```
Fitting 3 folds for each of 1536 candidates, totalling 4608 fits
[Parallel(n_jobs=-1)]: Done   1 jobs       | elapsed:    0.2s
[Parallel(n_jobs=-1)]: Done  50 jobs       | elapsed:    4.0s
[Parallel(n_jobs=-1)]: Done 200 jobs       | elapsed:   16.9s
[Parallel(n_jobs=-1)]: Done 450 jobs       | elapsed:   36.7s
[Parallel(n_jobs=-1)]: Done 800 jobs       | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 1250 jobs       | elapsed:  1.7min
[Parallel(n_jobs=-1)]: Done 1800 jobs       | elapsed:  2.5min
[Parallel(n_jobs=-1)]: Done 2450 jobs       | elapsed:  3.4min
[Parallel(n_jobs=-1)]: Done 3200 jobs       | elapsed:  4.4min
[Parallel(n_jobs=-1)]: Done 4050 jobs       | elapsed:  7.7min
[Parallel(n_jobs=-1)]: Done 4608 out of 4608 | elapsed:  8.5min
finished
Best score: 0.983
Best parameters set:
  clf__C: 10
  clf__penalty: 'l2'
  vect__max_df: 0.5
  vect__max_features: None
```

```
    vect__ngram_range: (1, 2)
    vect__norm: 'l2'
    vect__stop_words: None
    vect__use_idf: True
  Accuracy: 0.989956958393
  Precision: 0.988095238095
  Recall: 0.932584269663
```

Optimizing the values of the hyperparameters has improved our model's recall score on the `test` set.

# Multi-class classification

In the previous sections you learned to use logistic regression for binary classification. In many classification problems, however, there are more than two classes that are of interest. We might wish to predict the genres of songs from samples of audio, or classify images of galaxies by their types. The goal of **multi-class classification** is to assign an instance to one of the set of classes. scikit-learn uses a strategy called **one-vs.-all**, or **one-vs.-the-rest**, to support multi-class classification. One-vs.-all classification uses one binary classifier for each of the possible classes. The class that is predicted with the greatest confidence is assigned to the instance. `LogisticRegression` supports multi-class classification using the one-versus-all strategy out of the box. Let's use `LogisticRegression` for a multi-class classification problem.

Assume that you would like to watch a movie, but you have a strong aversion to watching bad movies. To inform your decision, you could read reviews of the movies you are considering, but unfortunately you also have a strong aversion to reading movie reviews. Let's use scikit-learn to find the movies with good reviews.

In this example, we will classify the sentiments of phrases taken from movie reviews in the Rotten Tomatoes data set. Each phrase can be classified as one of the following sentiments: negative, somewhat negative, neutral, somewhat positive, or positive. While the classes appear to be ordered, the explanatory variables that we will use do not always corroborate this order due to sarcasm, negation, and other linguistic phenomena. Instead, we will approach this problem as a multi-class classification task.

The data can be downloaded from `http://www.kaggle.com/c/sentiment-analysis-on-movie-reviews/data`. First, let's explore the data set using pandas. Note that the import and data-loading statements in the following snippet are required for the subsequent snippets:

```
>>> import pandas as pd
>>> df = pd.read_csv('movie-reviews/train.tsv', header=0,
delimiter='\t')
>>> print df.count()

PhraseId      156060
SentenceId    156060
Phrase        156060
Sentiment     156060
dtype: int64
```

The columns of the data set are tab delimited. The data set contains 1,56,060 instances.

```
>>> print df.head()

   PhraseId  SentenceId
Phrase  \
0         1           1  A series of escapades demonstrating the adage
...
1         2           1  A series of escapades demonstrating the adage
...
2         3           1                                             A
series
3         4           1
A
4         5           1
series

   Sentiment
0          1
1          2
2          2
3          2
4          2

[5 rows x 4 columns]
```

The `Sentiment` column contains the response variables. The `0` label corresponds to the sentiment `negative`, `1` corresponds to `somewhat negative`, and so on. The `Phrase` column contains the raw text. Each sentence from the movie reviews has been parsed into smaller phrases. We will not require the `PhraseId` and `SentenceId` columns in this example. Let's print some of the phrases and examine them:

```
>>> print df['Phrase'].head(10)

0    A series of escapades demonstrating the adage ...
1    A series of escapades demonstrating the adage ...
2                                           A series
3                                                  A
4                                             series
5    of escapades demonstrating the adage that what...
6                                                 of
7    escapades demonstrating the adage that what is...
8                                           escapades
9    demonstrating the adage that what is good for ...
Name: Phrase, dtype: object
```

Now let's examine the target classes:

```
>>> print df['Sentiment'].describe()

count    156060.000000
mean          2.063578
std           0.893832
min           0.000000
25%           2.000000
50%           2.000000
75%           3.000000
max           4.000000
Name: Sentiment, dtype: float64

>>> print df['Sentiment'].value_counts()

2    79582
3    32927
1    27273
4     9206
0     7072
```

```
dtype: int64

>>> print df['Sentiment'].value_counts()/df['Sentiment'].count()

2    0.509945
3    0.210989
1    0.174760
4    0.058990
0    0.045316
dtype: float64
```

The most common class, `Neutral`, includes more than 50 percent of the instances. Accuracy will not be an informative performance measure for this problem, as a degenerate classifier that predicts only `Neutral` can obtain an accuracy near 0.5. Approximately one quarter of the reviews are positive or somewhat positive, and approximately one fifth of the reviews are negative or somewhat negative. Let's train a classifier with scikit-learn:

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model.logistic import LogisticRegression
from sklearn.cross_validation import train_test_split
from sklearn.metrics.metrics import classification_report, accuracy_
score, confusion_matrix
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV

def main():
    pipeline = Pipeline([
        ('vect', TfidfVectorizer(stop_words='english')),
        ('clf', LogisticRegression())
    ])
    parameters = {
        'vect__max_df': (0.25, 0.5),
        'vect__ngram_range': ((1, 1), (1, 2)),
        'vect__use_idf': (True, False),
        'clf__C': (0.1, 1, 10),
    }
    df = pd.read_csv('data/train.tsv', header=0, delimiter='\t')
    X, y = df['Phrase'], df['Sentiment'].as_matrix()
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_
size=0.5)
    grid_search = GridSearchCV(pipeline, parameters, n_jobs=3,
verbose=1, scoring='accuracy')
```

```
    grid_search.fit(X_train, y_train)
    print 'Best score: %0.3f' % grid_search.best_score_
    print 'Best parameters set:'
    best_parameters = grid_search.best_estimator_.get_params()
    for param_name in sorted(parameters.keys()):
        print '\t%s: %r' % (param_name, best_parameters[param_name])

if __name__ == '__main__':
    main()
```

The following is the output of the script:

```
Fitting 3 folds for each of 24 candidates, totalling 72 fits
[Parallel(n_jobs=3)]: Done   1 jobs       | elapsed:    3.3s
[Parallel(n_jobs=3)]: Done  50 jobs       | elapsed:  1.1min
[Parallel(n_jobs=3)]: Done  68 out of  72 | elapsed:  1.9min
remaining:    6.8s
[Parallel(n_jobs=3)]: Done  72 out of  72 | elapsed:  2.1min finished
Best score: 0.620
Best parameters set:
  clf__C: 10
  vect__max_df: 0.25
  vect__ngram_range: (1, 2)
  vect__use_idf: False
```

# Multi-class classification performance metrics

As with binary classification, confusion matrices are useful for visualizing the types of errors made by the classifier. Precision, recall, and F1 score can be computed for each of the classes, and accuracy for all of the predictions can also be calculated. Let's evaluate our classifier's predictions. The following snippet continues the previous example:

```
    predictions = grid_search.predict(X_test)
    print 'Accuracy:', accuracy_score(y_test, predictions)
    print 'Confusion Matrix:', confusion_matrix(y_test, predictions)
    print 'Classification Report:', classification_report(y_test,
predictions)
```

The following will be appended to the output:

```
Accuracy: 0.636370626682
Confusion Matrix: [[ 1129  1679   634    64     9]
 [  917  6121  6084   505    35]
 [  229  3091 32688  3614   166]
 [   34   408  6734  8068  1299]
 [    5    35   494  2338  1650]]
Classification Report:              precision    recall  f1-score
support

          0       0.49      0.32      0.39      3515
          1       0.54      0.45      0.49     13662
          2       0.70      0.82      0.76     39788
          3       0.55      0.49      0.52     16543
          4       0.52      0.36      0.43      4522

avg / total       0.62      0.64      0.62     78030
```

First, we make predictions using the best parameter set found by using grid searching. While our classifier is an improvement over the baseline classifier, it frequently mistakes `Somewhat Positive` and `Somewhat Negative` for `Neutral`.

# Multi-label classification and problem transformation

In the previous sections, we discussed binary classification, in which each instance must be assigned to one of the two classes, and multi-class classification, in which each instance must be assigned to one of the set of classes. The final type of classification problem that we will discuss is multi-label classification, in which each instance can be assigned a subset of the set of classes. Examples of multi-label classification include assigning tags to messages posted on a forum, and classifying the objects present in an image. There are two groups of approaches for multi-label classification.

**Problem transformation** methods are techniques that cast the original multi-label problem as a set of single-label classification problems. The first problem transformation method that we will review converts each set of labels encountered in the training data to a single label. For example, consider a multi-label classification problem in which news articles must be assigned to one or more categories from a set. The following training data contains seven articles that can pertain to one or more of the five categories.

| | **Categories** | | | | |
|---|---|---|---|---|---|
| **Instance** | **Local** | **US** | **Business** | **Science and Technology** | **Sports** |
| 1 | ✔ | ✔ | | | |
| 2 | ✔ | | ✔ | | |
| 3 | | | ✔ | ✔ | |
| 4 | | | | | ✔ |
| 5 | ✔ | | | | |
| 6 | | | ✔ | | |
| 7 | | ✔ | | ✔ | |

Transforming the problem into a single-label classification task using the power set of labels seen in the training data results in the following training data. Previously, the first instance was classified as `Local` and `US`. Now it has a single label, `Local ∧ US`.

| | **Category** | | | | | | |
|---|---|---|---|---|---|---|---|
| **Instance** | **Local** | **Local ∧ US** | **Business** | **Local ∧ Business** | **US ∧ Science and Technology** | **Business ∧ Science and Technology** | **Sports** |
| 1 | | ✔ | | | | | |
| 2 | | | | ✔ | | | |
| 3 | | | | | | ✔ | |
| 4 | | | | | | | ✔ |
| 5 | ✔ | | | | | | |
| 6 | | | ✔ | | | | |
| 7 | | | | | ✔ | | |

The multi-label classification problem that had five classes is now a multi-class classification problem with seven classes. While the power set problem transformation is intuitive, increasing the number of classes is frequently impractical; this transformation can produce many new labels that correspond to only a few training instances. Furthermore, the classifier can only predict combinations of labels that were seen in the training data.

| | **Category** | | | **Category** | |
|---|---|---|---|---|---|
| **Instance** | **Local** | **¬Local** | **Instance** | **Business** | **¬Business** |
| 1 | ✔ | | 1 | | ✔ |
| 2 | ✔ | | 2 | ✔ | |
| 3 | | ✔ | 3 | ✔ | |
| 4 | | ✔ | 4 | | ✔ |
| 5 | ✔ | | 5 | | ✔ |
| 6 | | ✔ | 6 | ✔ | |
| 7 | | ✔ | 7 | | ✔ |
| | | | | | |
| | **Category** | | | **Category** | |
| Instance | US | ¬US | Instance | US | ¬US |
| 1 | ✔ | | 1 | ✔ | |
| 2 | ✔ | | 2 | ✔ | |
| 3 | | ✔ | 3 | | ✔ |
| 4 | | ✔ | 4 | | ✔ |
| 5 | | ✔ | 5 | | ✔ |
| 6 | | ✔ | 6 | | ✔ |
| 7 | ✔ | | 7 | ✔ | |
| | | | | | |
| | **Category** | | | **Category** | |
| Instance | Sci. and Tech. | ¬Sci. and Tech. | Instance | Sports | ¬Sports |
| 1 | | ✔ | 1 | | ✔ |
| 2 | | ✔ | 2 | | ✔ |
| 3 | ✔ | | 3 | | ✔ |
| 4 | | ✔ | 4 | ✔ | |
| 5 | | ✔ | 5 | | ✔ |
| 6 | | ✔ | 6 | | ✔ |

A second problem transformation is to train one binary classifier for each of the labels in the training set. Each classifier predicts whether or not the instance belongs to one label. Our example would require five binary classifiers; the first classifier would predict whether or not an instance should be classified as `Local`, the second classifier would predict whether or not an instance should be classified as `US`, and so on. The final prediction is the union of the predictions from all of the binary classifiers. The transformed training data is shown in the previous figure. This problem transformation ensures that the single-label problems will have the same number of training examples as the multilabel problem, but ignores relationships between the labels.

# Multi-label classification performance metrics

Multi-label classification problems must be assessed using different performance measures than single-label classification problems. Two of the most common performance metrics are **Hamming loss** and **Jaccard similarity**. Hamming loss is the average fraction of incorrect labels. Note that Hamming loss is a loss function, and that the perfect score is zero. Jaccard similarity, or the Jaccard index, is the size of the intersection of the predicted labels and the true labels divided by the size of the union of the predicted and true labels. It ranges from zero to one, and one is the perfect score. Jaccard similarity is calculated by the following equation:

$$J\left(\text{Predicted},\text{True}\right)=\frac{\left|\text{Predicted}\cap\text{True}\right|}{\left|\text{Predicted}\cup\text{True}\right|}$$

```
>>> import numpy as np
>>> from sklearn.metrics import hamming_loss
>>> print hamming_loss(np.array([[0.0, 1.0], [1.0, 1.0]]),
np.array([[0.0, 1.0], [1.0, 1.0]]))
0.0
>>> print hamming_loss(np.array([[0.0, 1.0], [1.0, 1.0]]),
np.array([[1.0, 1.0], [1.0, 1.0]]))
0.25
>>> print hamming_loss(np.array([[0.0, 1.0], [1.0, 1.0]]),
np.array([[1.0, 1.0], [0.0, 1.0]]))
0.5
```

```
>>> print jaccard_similarity_score(np.array([[0.0, 1.0], [1.0, 1.0]]),
np.array([[0.0, 1.0], [1.0, 1.0]]))
1.0
>>> print jaccard_similarity_score(np.array([[0.0, 1.0], [1.0, 1.0]]),
np.array([[1.0, 1.0], [1.0, 1.0]]))
0.75
>>> print jaccard_similarity_score(np.array([[0.0, 1.0], [1.0, 1.0]]),
np.array([[1.0, 1.0], [0.0, 1.0]]))
0.5
```

# Summary

In this chapter we discussed generalized linear models, which extend ordinary linear regression to support response variables with non-normal distributions. Generalized linear models use a link function to relate a linear combination of the explanatory variables to the response variable; unlike ordinary linear regression, the relationship does not need to be linear. In particular, we examined the logistic link function, a sigmoid function that returns a value between zero and one for any real number.

We discussed logistic regression, a generalized linear model that uses the logistic link function to relate explanatory variables to a Bernoulli-distributed response variable. Logistic regression can be used for binary classification, a task in which an instance must be assigned to one of the two classes; we used logistic regression to classify spam and ham SMS messages. We then discussed multi-class classification, a task in which each instance must be assigned one label from a set of labels. We used the one-vs.-all strategy to classify the sentiments of movie reviews. Finally, we discussed multi-label classification, in which instances must be assigned a subset of a set of labels. Having completed our discussion of regression and classification with generalized linear models, we will introduce a non-linear model for regression and classification called the decision tree in the next chapter.

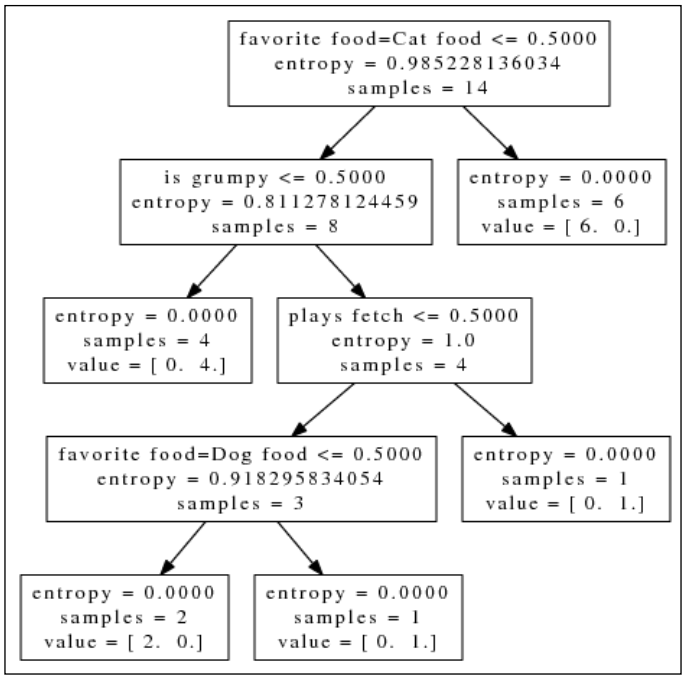
# 5
# Nonlinear Classification and Regression with Decision Trees

In the previous chapters we discussed generalized linear models, which relate a linear combination of explanatory variables to one or more response variables using a link function. You learned to use multiple linear regression to solve regression problems, and we used logistic regression for classification tasks. In this chapter we will discuss a simple, nonlinear model for classification and regression tasks: the decision tree. We'll use decision trees to build an ad blocker that can learn to classify images on a web page as banner advertisements or page content. Finally, we will introduce ensemble learning methods, which combine a set of models to produce an estimator with better predictive performance than any of its component estimators.

## Decision trees

Decision trees are tree-like graphs that model a decision. They are analogous to the parlor game Twenty Questions. In Twenty Questions, one player, called the answerer, chooses an object but does not reveal the object to the other players, who are called questioners. The object should be a common noun, such as "guitar" or "sandwich", but not "1969 Gibson Les Paul Custom" or "North Carolina". The questioners must guess the object by asking as many as twenty questions that can be answered with `yes`, `no`, or `maybe`. An intuitive strategy for questioners is to ask questions of increasing specificity; asking "*is it a musical instrument?*" as the first question will not efficiently reduce the number of possibilities. The branches of a decision tree specify the shortest sequences of explanatory variables that can be examined in order to estimate the value of a response variable. To continue the analogy, in Twenty Questions the questioner and the answerers all have knowledge of the training data, but only the answerer knows the values of the features for the test instance.

Decision trees are commonly learned by recursively splitting the set of training instances into subsets based on the instances' values for the explanatory variables. The following diagram depicts a decision tree that we will look at in more detail later in the chapter.




Represented by boxes, the interior nodes of the decision tree test explanatory variables. These nodes are connected by edges that specify the possible outcomes of the tests. The training instances are divided into subsets based on the outcomes of the tests. For example, a node might test whether or not the value of an explanatory variable exceeds a threshold. The instances that pass the test will follow an edge to the node's right child, and the instances that fail the test will follow an edge to the node's left child. The children nodes similarly test their subsets of the training instances until a stopping criterion is satisfied. In classification tasks, the leaf nodes of the decision tree represent classes. In regression tasks, the values of the response variable for the instances contained in a leaf node may be averaged to produce the estimate for the response variable. After the decision tree has been constructed, making a prediction for a test instance requires only following the edges until a leaf node is reached.

# Training decision trees

Let's create a decision tree using an algorithm called **Iterative Dichotomiser 3** (**ID3**). Invented by Ross Quinlan, ID3 was one of the first algorithms used to train decision trees. Assume that you have to classify animals as cats or dogs. Unfortunately, you cannot observe the animals directly and must use only a few attributes of the animals to make your decision. For each animal, you are told whether or not it likes to play fetch, whether or not it is frequently grumpy, and its favorite of three types of food.

To classify new animals, the decision tree will examine an explanatory variable at each node. The edge it follows to the next node will depend on the outcome of the test. For example, the first node might ask whether or not the animal likes to play fetch. If the animal does, we will follow the edge to the left child node; if not, we will follow the edge to the right child node. Eventually an edge will connect to a leaf node that indicates whether the animal is a cat or a dog.

The following fourteen instances comprise our training data:

| Training instance | Plays fetch | Is grumpy | Favorite food | Species |
|---|---|---|---|---|
| 1 | Yes | No | Bacon | Dog |
| 2 | No | Yes | Dog Food | Dog |
| 3 | No | Yes | Cat food | Cat |
| 4 | No | Yes | Bacon | Cat |
| 5 | No | No | Cat food | Cat |
| 6 | No | Yes | Bacon | Cat |
| 7 | No | Yes | Cat Food | Cat |
| 8 | No | No | Dog Food | Dog |
| 9 | No | Yes | Cat food | Cat |
| 10 | Yes | No | Dog Food | Dog |
| 11 | Yes | No | Bacon | Dog |
| 12 | No | No | Cat food | Cat |
| 13 | Yes | Yes | Cat food | Cat |
| 14 | Yes | Yes | Bacon | Dog |

From this data we can see that cats are generally grumpier than the dogs. Most dogs play fetch and most cats refuse. Dogs prefer dog food and bacon, whereas cats only like cat food and bacon. The `is grumpy` and `plays fetch` explanatory variables can be easily converted to binary-valued features. The `favorite food` explanatory variable is a categorical variable that has three possible values; we will one-hot encode it. Recall from *Chapter 3*, *Feature Extraction and Preprocessing*, that one-hot encoding represents a categorical variable with as many binary-valued features as there are values for variable. Representing the categorical variable with a single integer-valued feature will encode an artificial order to its values. Since `favorite food` has three possible states, we will represent it with three binary-valued features. From this table, we can manually construct classification rules. For example, an animal that is grumpy and likes cat food must be a cat, while an animal that plays fetch and likes bacon must be a dog. Constructing these classification rules by hand for even a small data set is cumbersome. Instead, we will learn these rules by creating a decision tree.

# Selecting the questions

Like Twenty Questions, the decision tree will estimate the value of the response variable by testing the values of a sequence of explanatory variables. Which explanatory variable should be tested first? Intuitively, a test that produces subsets that contain all cats or all dogs is better than a test that produces subsets that still contain both cats and dogs. If the members of a subset are of different classes, we are still uncertain about how to classify the instance. We should also avoid creating tests that separate only a single cat or dog from the others; such tests are analogous to asking specific questions in the first few rounds of Twenty Questions. More formally, these tests can infrequently classify an instance and might not reduce our uncertainty. The tests that reduce our uncertainty about the classification the most are the best. We can quantify the amount of uncertainty using a measure called **entropy**.

Measured in bits, entropy quantifies the amount of uncertainty in a variable. Entropy is given by the following equation, where $n$ is the number of outcomes and $P\left(x_i\right)$ is the probability of the outcome $i$. Common values for $b$ are `2`, $e$, and `10`. Because the log of a number less than one will be negative, the entire sum is negated to return a positive value.

$$H\left(X\right) = -\sum_{i=1}^{n} P\left(x_i\right) \log_b P\left(x_i\right)$$

For example, a single toss of a fair coin has only two outcomes: heads and tails. The probability that the coin will land on heads is 0.5, and the probability that it will land on tails is 0.5. The entropy of the coin toss is equal to the following:

$$H(X) = -(0.5\log_2 0.5 + 0.5\log_2 0.5) = 1.0$$

That is, only one bit is required to represent the two equally probable outcomes, heads and tails. Two tosses of a fair coin can result in four possible outcomes: heads and heads, heads and tails, tails and heads, and tails and tails. The probability of each outcome is *0.5 x 0.5 = 0.25*. The entropy of two tosses is equal to the following:

$$H(X) = -(0.25\log_2 0.25 + 0.25\log_2 0.25 + 0.25\log_2 0.25 + 0.25\log_2 0.25) = 2.0$$

If the coin has the same face on both sides, the variable representing its outcome has 0 bits of entropy; that is, we are always certain of the outcome and the variable will never represent new information. Entropy can also be represented as a fraction of a bit. For example, an unfair coin has two different faces, but is weighted such that the faces are not equally likely to land in a toss. Assume that the probability that an unfair coin will land on heads is 0.8, and the probability that it will land on tails is 0.2. The entropy of a single toss of this coin is equal to the following:

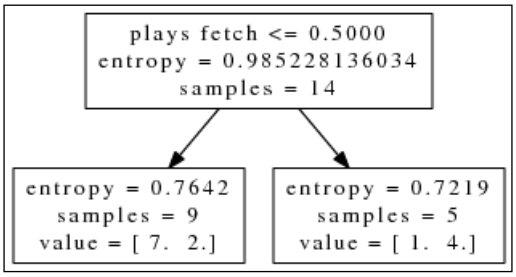
$$H(X) = -(0.9\log_2 0.9 + 0.2\log_2 0.2) = 0.7219280948873623$$

The outcome of a single toss of an unfair coin can have a fraction of one bit of entropy. There are two possible outcomes of the toss, but we are not totally uncertain since one outcome is more frequent.

Let's calculate the entropy of classifying an unknown animal. If an equal number of dogs and cats comprise our animal classification training data and we do not know anything else about the animal, the entropy of the decision is equal to one. All we know is that the animal could be either a cat or a dog; like the fair coin toss, both outcomes are equally likely. Our training data, however, contains six dogs and eight cats. If we do not know anything else about the unknown animal, the entropy of the decision is given by the following:

$$H(X) = -\left(\frac{6}{14}\log_2\frac{6}{14}\log_2 + \frac{8}{14}\log_2\frac{8}{14}\right) = 0.985228136.342516$$

Since cats are more common, we are less uncertain about the outcome. Now let's find the explanatory variable that will be most helpful in classifying the animal; that is, let's find the explanatory variable that reduces the entropy the most. We can test the `plays fetch` explanatory variable and divide the training instances into animals that play fetch and animals that don't. This produces the two following subsets:



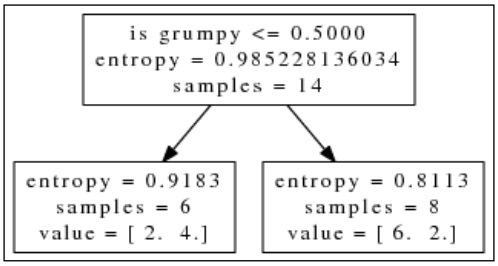

Decision trees are often visualized as diagrams that are similar to flowcharts. The top box of the previous diagram is the root node; it contains all of our training instances and specifies the explanatory variable that will be tested. At the root node we have not eliminated any instances from the training set and the entropy is equal to approximately 0.985. The root node tests the `plays fetch` explanatory variable. Recall that we converted this Boolean explanatory variable to a binary-valued feature. Training instances for which `plays fetch` is equal to zero follow the edge to the root's left child, and training instances for animals that do play fetch follow the edge to the root's right child node. The left child node contains a subset of the training data with seven cats and two dogs that do not like to play fetch. The entropy at this node is given by the following:

$$H(X) = -\left(\frac{2}{9}\log_2\frac{2}{9} + \frac{7}{9}\log_2\frac{7}{9}\right) = 0.7642045065086203$$

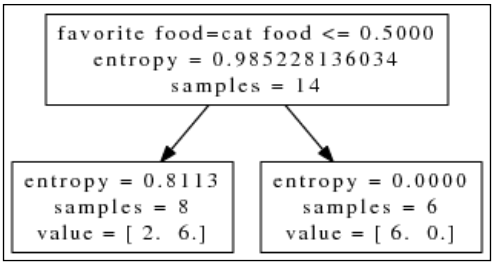
The right child contains a subset with one cat and four dogs that do like to play fetch. The entropy at this node is given by the following:

$$H(X) = -\left(\frac{1}{5}\log_2\frac{1}{5} + \frac{4}{5}\log_2\frac{4}{5}\right) = 0.7219280948873623$$

Instead of testing the `plays fetch` explanatory variable, we could test the `is grumpy` explanatory variable. This test produces the following tree. As with the previous tree, instances that fail the test follow the left edge, and instances that pass the test follow the right edge.




We could also divide the instances into animals that prefer cat food and animals that don't to produce the following tree:




# Information gain

Testing for the animals that prefer cat food resulted in one subset with six cats, zero dogs, and 0 bits of entropy and another subset with two cats, six dogs, and 0.811 bits of entropy. How can we measure which of these tests reduced our uncertainty about the classification the most? Averaging the entropies of the subsets may seem to be an appropriate measure of the reduction in entropy. In this example, the subsets produced by the cat food test have the lowest average entropy. Intuitively, this test seems to be effective, as we can use it to classify almost half of the training instances. However, selecting the test that produces the subsets with the lowest average entropy can produce a suboptimal tree. For example, imagine a test that produced one subset with two dogs and no cats and another subset with four dogs and eight cats. The entropy of the first subset is equal to the following (note that the second term is omitted because $\log_2 0$ is undefined):

$$H(X) = -\left(\frac{2}{2}\log_2\frac{2}{2}\right) = 0.0$$

The entropy of the second subset is equal to the following:

$$H(X) = -\left(\frac{4}{12}\log_2\frac{4}{12} + \frac{8}{12}\log_2\frac{8}{12}\right) = 0.9182958340544896$$

The average of these subsets' entropies is only 0.459, but the subset containing most of the instances has almost one bit of entropy. This is analogous to asking specific questions early in Twenty Questions; we could get lucky and win within the first few attempts, but it is more likely that we will squander our questions without eliminating many possibilities. Instead, we will measure the reduction in entropy using a metric called **information gain**. Calculated with the following equation, information gain is the difference between the entropy of the parent node, $H(T)$, and the weighted average of the children nodes' entropies. $T$ is the set of instances, and $a$ is the explanatory variable under test. $x_a \in vals(a)$ is the value of attribute $a$ for instance $x$. $\{x \in T \mid x_a = v\}$ is the number of instances for which attribute $a$ is equal to the value $v$. $H(\{x \in T \mid x_a = v\})$ is the entropy of the subset of instances for which the value of the explanatory variable $a$ is $v$.

$$IG(T,a) = H(T) - \sum_{v \in vals(a)} \frac{|\{x \in T \mid x_a = v\}|}{|T|} H(\{x \in T \mid x_a = v\})$$

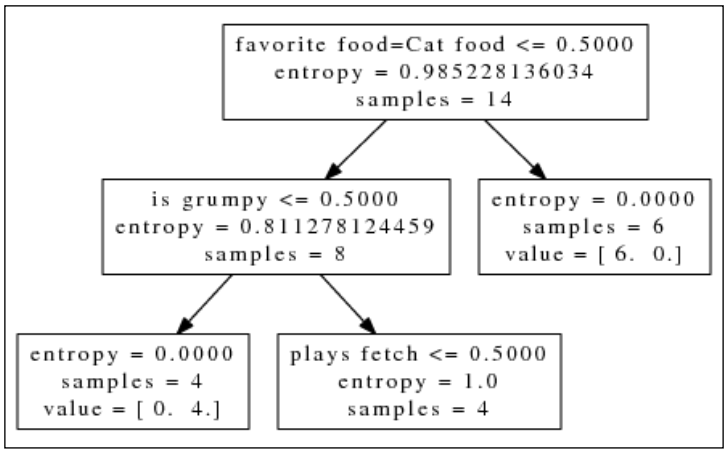
The following table contains the information gains for all of the tests. In this case, the cat food test is still the best, as it increases the information gain the most.

| **Test** | **Parent's entropy** | **Child's entropy** | **Child's entropy** | **Weighted average** | **IG** |
|---|---|---|---|---|---|
| **plays fetch?** | 0.9852 | 0.7642 | 0.7219 | 0.7490 * 9/14 + 0.7219 * 5/14 = 0.7491 | 0.2361 |
| **is grumpy?** | 0.9852 | 0.9183 | 0.8113 | 0.9183 * 6/14 + 0.8113 * 8/14 = 0.85710.8572 | 0.1280 |
| **favorite food = cat food** | 0.9852 | 0.8113 | 0 | 0.8113 * 8 /14 + 0.0 * 6/14 = 0.4636 | 0.5216 |
| **favorite food = dog food** | 0.9852 | 0.8454 | 0 | 0.8454 * 11/14 + 0.0 * 3/14 = 0.6642 | 0.3210 |
| **favorite food = bacon** | 0.9852 | 0.9183 | 0.971 | 0.9183 * 9/14 + 0.9710 * 5/14 = 0.9371 | 0.0481 |

Now let's add another node to the tree. One of the child nodes produced by the test is a leaf node that contains only cats. The other node still contains two cats and six dogs. We will add a test to this node. Which of the remaining explanatory variables reduces our uncertainty the most? The following table contains the information gains for all of the possible tests:

| **Test** | **Parent's entropy** | **Child's entropy** | **Child's entropy** | **Weighted average** | **IG** |
|---|---|---|---|---|---|
| **plays fetch?** | 0.8113 | 1 | 0 | 1.0 * 4/8 + 0 * 4/8 = 0.5 | 0.3113 |
| **is grumpy?** | 0.8113 | 0 | 1 | 0.0 * 4/8 + 1 * 4/8 = 0.5 | 0.3113 |
| **favorite food=dog food** | 0.8113 | 0.9710 | 0 | 0.9710 * 5/8 + 0.0 * 3/8 = 0.6069 | 0.2044 |
| **favorite food=bacon** | 0.8113 | 0 | 0.9710 | 0.0 * 3/8 + 0.9710 * 5/8 = 0.6069 | 0.2044 |

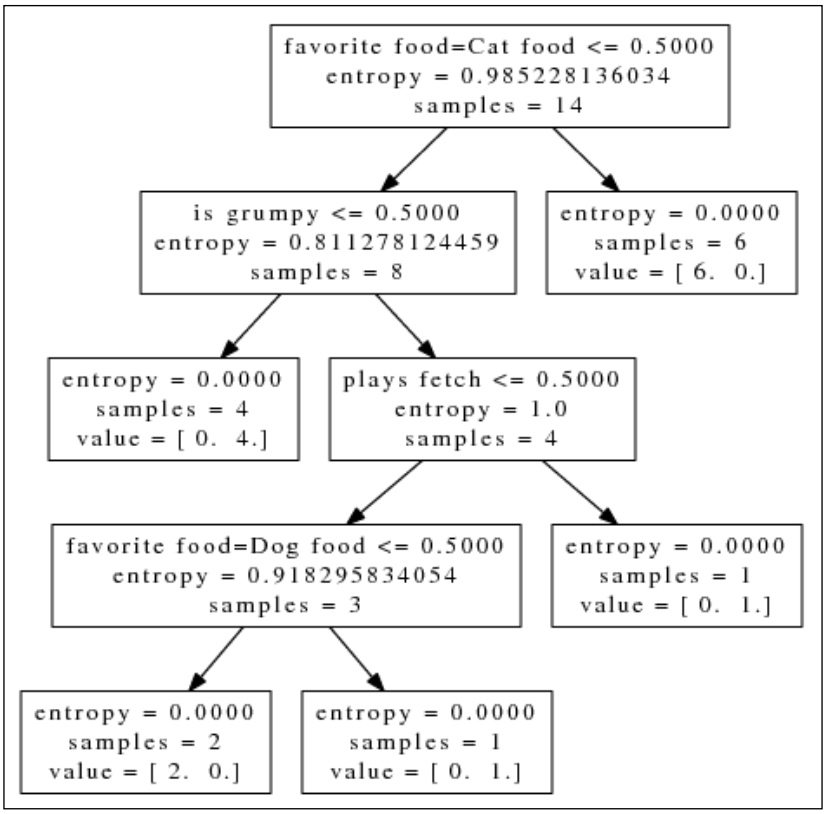
All of the tests produce subsets with 0 bits of entropy, but the `is grumpy` and `plays fetch` tests produce the greatest information gain. ID3 breaks ties by selecting one of the best tests arbitrarily. We will select the `is grumpy` test, which splits its parent's eight instances into a leaf node containing four dogs and a node containing two cats and two dogs. The following is a diagram of the current tree:



We will now select another explanatory variable to test the child node's four instances. The remaining tests, `favorite food=bacon`, `favorite food=dog food`, and `plays fetch`, all produce a leaf node containing one dog or cat and a node containing the remaining animals. The remaining tests produce equal information gains, as shown in the following table:

| Test | Parent's Entropy | Child Entropy | Child Entropy | Weighted Average | Information Gain |
|---|---|---|---|---|---|
| **plays fetch?** | 1 | 0.9183 | 0 | 0.688725 | 0.311275 |
| **favorite food=dog food** | 1 | 0.9183 | 0 | 0.688725 | 0.311275 |
| **favorite food=bacon** | 1 | 0 | 0.9183 | 0.688725 | 0.311275 |

We will arbitrarily select the `plays fetch` test to produce a leaf node containing one dog and a node containing two cats and a dog. Two explanatory variables remain; we can test for animals that like bacon, or we can test for animals that like dog food. Both of the tests will produce the same subsets and create a leaf node containing one dog and a leaf node containing two cats. We will arbitrarily choose to test for animals that like dog food. The following is a diagram of the completed decision tree:




Let's classify some animals from the following test data:

| Testing instance | Plays fetch | Is grumpy | Favorite food | Species |
|---|---|---|---|---|
| 1 | Yes | No | Bacon | Dog |
| 2 | Yes | Yes | Dog Food | Dog |
| 3 | No | Yes | Dog Food | Cat |
| 4 | No | Yes | Bacon | Cat |
| 5 | No | No | Cat food | Cat |

Let's classify the first animal, which likes to plays fetch, is infrequently grumpy, and loves bacon. We will follow the edge to the root node's left child since the animal's favorite food is not cat food. The animal is not grumpy, so we will follow the edge to the second-level node's left child. This is a leaf node containing only dogs; we have correctly classified this instance. To classify the third test instance as a cat, we follow the edge to the root node's left child, follow the edge to the second-level node's right child, follow the edge to the third-level node's left child, and finally follow the edge to the fourth-level node's right child.

Congratulations! You've constructed a decision tree using the ID3 algorithm. Other algorithms can be used to train decision trees. **C4.5** is a modified version of ID3 that can be used with continuous explanatory variables and can accommodate missing values for features. C4.5 also can **prune** trees. Pruning reduces the size of a tree by replacing branches that classify few instances with leaf nodes. Used by scikit-learn's implementation of decision trees, **CART** is another learning algorithm that supports pruning.

# Gini impurity

In the previous section, we built a decision tree by creating nodes that produced the greatest information gain. Another common heuristic for learning decision trees is **Gini impurity**, which measures the proportions of classes in a set. Gini impurity is given by the following equation, where $j$ is the number of classes, $t$ is the subset of instances for the node, and $P(i \mid t)$ is the probability of selecting an element of class $i$ from the node's subset:

$$Gini(t) = 1 - \sum_{i=1}^{j} P(i \mid t)^2$$

Intuitively, Gini impurity is zero when all of the elements of the set are the same class, as the probability of selecting an element of that class is equal to one. Like entropy, Gini impurity is greatest when each class has an equal probability of being selected. The maximum value of Gini impurity depends on the number of possible classes, and it is given by the following equation:

$$Gini_{\max} = 1 - \frac{1}{n}$$

Our problem has two classes, so the maximum value of the Gini impurity measure will be equal to one half. scikit-learn supports learning decision trees using both information gain and Gini impurity. There are no firm rules to help you decide when to use one criterion or the other; in practice, they often produce similar results. As with many decisions in machine learning, it is best to compare the performances of models trained using both options.

# Decision trees with scikit-learn

Let's use decision trees to create software that can block banner ads on web pages. This program will predict whether each of the images on a web page is an advertisement or article content. Images that are classified as being advertisements could then be hidden using Cascading Style Sheets. We will train a decision tree classifier using the *Internet Advertisements Data Set* from `http://archive.ics.uci.edu/ml/datasets/Internet+Advertisements`, which contains data for 3,279 images. The proportions of the classes are skewed; 459 of the images are advertisements and 2,820 are content. Decision tree learning algorithms can produce biased trees from data with unbalanced class proportions; we will evaluate a model on the unaltered data set before deciding if it is worth balancing the training data by over- or under-sampling instances. The explanatory variables are the dimensions of the image, words from the containing page's URL, words from the image's URL, the image's alt text, the image's anchor text, and a window of words surrounding the image tag. The response variable is the image's class. The explanatory variables have already been transformed into feature representations. The first three features are real numbers that encode the width, height, and aspect ratio of the images. The remaining features encode binary term frequencies for the text variables. In the following sample, we will grid search for the hyperparameter values that produce the decision tree with the greatest accuracy, and then evaluate the tree's performance on a test set:

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.cross_validation import train_test_split
from sklearn.metrics import classification_report
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV
```

First we read the `.csv` file using pandas. The `.csv` does not have a header row, so we split the last column containing the response variable's values from the features using its index:

```
if __name__ == '__main__':
    df = pd.read_csv('data/ad.data', header=None)
    explanatory_variable_columns = set(df.columns.values)
    response_variable_column = df[len(df.columns.values)-1]
    # The last column describes the targets
    explanatory_variable_columns.remove(len(df.columns.values)-1)

    y = [1 if e == 'ad.' else 0 for e in response_variable_column]
    X = df[list(explanatory_variable_columns)]
```

We encoded the advertisements as the positive class and the content as the negative class. More than one quarter of the instances are missing at least one of the values for the image's dimensions. These missing values are marked by whitespace and a question mark. We replaced the missing values with negative one, but we could have imputed the missing values; for instance, we could have replaced the missing height values with the average height value:

```
    X.replace(to_replace=' *\?', value=-1, regex=True, inplace=True)
```

We then split the data into training and test sets:

```
    X_train, X_test, y_train, y_test = train_test_split(X, y)
```

We created a pipeline and an instance of `DecisionTreeClassifier`. Then, we set the `criterion` keyword argument to `entropy` to build the tree using the information gain heuristic:

```
    pipeline = Pipeline([
        ('clf', DecisionTreeClassifier(criterion='entropy'))
    ])
```

Next, we specified the hyperparameter space for the grid search:

```
    parameters = {
        'clf__max_depth': (150, 155, 160),
        'clf__min_samples_split': (1, 2, 3),
        'clf__min_samples_leaf': (1, 2, 3)
    }
```

We set `GridSearchCV()` to maximize the model's F1 score:

```
    grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1,
verbose=1, scoring='f1')
    grid_search.fit(X_train, y_train)
    print 'Best score: %0.3f' % grid_search.best_score_
    print 'Best parameters set:'
    best_parameters = grid_search.best_estimator_.get_params()
    for param_name in sorted(parameters.keys()):
        print '\t%s: %r' % (param_name, best_parameters[param_name])

    predictions = grid_search.predict(X_test)
    print classification_report(y_test, predictions)

Fitting 3 folds for each of 27 candidates, totalling 81 fits
[Parallel(n_jobs=-1)]: Done   1 jobs       | elapsed:    1.7s
[Parallel(n_jobs=-1)]: Done  50 jobs       | elapsed:   15.0s
[Parallel(n_jobs=-1)]: Done  71 out of  81 | elapsed:   20.7s
remaining:    2.9s
[Parallel(n_jobs=-1)]: Done  81 out of  81 | elapsed:   23.3s finished
Best score: 0.878
Best parameters set:
   clf__max_depth: 155
   clf__min_samples_leaf: 2
   clf__min_samples_split: 1
             precision    recall  f1-score   support

          0       0.97      0.99      0.98       710
          1       0.92      0.81      0.86       110

avg / total       0.96      0.96      0.96       820
```

The classifier detected more than 80 percent of the ads in the test set, and approximately 92 percent of the images that it predicted were ads were truly ads. Overall, the performance is promising; in following sections, we will try to modify our model to improve its performance.

# Tree ensembles

**Ensemble learning** methods combine a set of models to produce an estimator that has better predictive performance than its individual components. A **random forest** is a collection of decision trees that have been trained on randomly selected subsets of the training instances and explanatory variables. Random forests usually make predictions by returning the mode or mean of the predictions of their constituent trees; scikit-learn's implementations return the mean of the trees' predictions. Random forests are less prone to overfitting than decision trees because no single tree can learn from all of the instances and explanatory variables; no single tree can memorize all of the noise in the representation.

Let's update our ad blocker's classifier to use a random forest. It is simple to replace the `DecisionTreeClassifier` using scikit-learn's API; we simply replace the object with an instance of `RandomForestClassifier`. Like the previous example, we will grid search to find the values of the hyperparameters that produce the random forest with the best predictive performance.

First, import the `RandomForestClassifier` class from the `ensemble` module:

```
from sklearn.ensemble import RandomForestClassifier
```

Next, replace the `DecisionTreeClassifier` in the `pipeline` with an instance of `RandomForestClassifier` and update the hyperparameter space:

```
pipeline = Pipeline([
    ('clf', RandomForestClassifier(criterion='entropy'))
])
parameters = {
    'clf__n_estimators': (5, 10, 20, 50),
    'clf__max_depth': (50, 150, 250),
    'clf__min_samples_split': (1, 2, 3),
    'clf__min_samples_leaf': (1, 2, 3)
}
```

The output is as follows:

```
Fitting 3 folds for each of 108 candidates, totalling 324 fits
[Parallel(n_jobs=-1)]: Done   1 jobs       | elapsed:    1.1s
[Parallel(n_jobs=-1)]: Done  50 jobs       | elapsed:   17.4s
[Parallel(n_jobs=-1)]: Done 200 jobs       | elapsed:  1.0min
[Parallel(n_jobs=-1)]: Done 324 out of 324 | elapsed:  1.6min finished
Best score: 0.936
```

```
Best parameters set:
   clf__max_depth: 250
   clf__min_samples_leaf: 1
   clf__min_samples_split: 3
   clf__n_estimators: 20
             precision    recall  f1-score   support

          0       0.97      1.00      0.98       705
          1       0.97      0.83      0.90       115

avg / total       0.97      0.97      0.97       820
```

Replacing the single decision tree with a random forest resulted in a significant reduction of the error rate; the random forest improves the precision and recall for ads to 0.97 and 0.83.

# The advantages and disadvantages of decision trees

The compromises associated with using decision trees are different than those of the other models we discussed. Decision trees are easy to use. Unlike many learning algorithms, decision trees do not require the data to have zero mean and unit variance. While decision trees can tolerate missing values for explanatory variables, scikit-learn's current implementation cannot. Decision trees can even learn to ignore explanatory variables that are not relevant to the task.

Small decision trees can be easy to interpret and visualize with the `export_graphviz` function from scikit-learn's `tree` module. The branches of a decision tree are conjunctions of logical predicates, and they are easily visualized as flowcharts. Decision trees support multioutput tasks, and a single decision tree can be used for multiclass classification without employing a strategy like one-versus-all.

Like the other models we discussed, decision trees are **eager learners**. Eager learners must build an input-independent model from the training data before they can be used to estimate the values of test instances, but can predict relatively quickly once the model has been built. In contrast, **lazy learners** such as the k-nearest neighbors algorithm defer all generalization until they must make a prediction. Lazy learners do not spend time training, but often predict slowly compared to eager learners.

Decision trees are more prone to overfitting than many of the models we discussed, as their learning algorithms can produce large, complicated decision trees that perfectly model every training instance but fail to generalize the real relationship. Several techniques can mitigate over-fitting in decision trees. **Pruning** is a common strategy that removes some of the tallest nodes and leaves of a decision tree, but it is not currently implemented in scikit-learn. However, similar effects can be achieved by setting a maximum depth for the tree or by creating child nodes only when the number of training instances they will contain exceeds a threshold. The `DecisionTreeClassifier` and `DecisionTreeRegressor` classes provide keyword arguments to set these constraints. Creating a random forest can also reduce over-fitting.

Efficient decision tree learning algorithms like ID3 are **greedy**. They learn efficiently by making locally optimal decisions, but are not guaranteed to produce the globally optimal tree. ID3 constructs a tree by selecting a sequence of explanatory variables to test. Each explanatory variable is selected because it reduces the uncertainty in the node more than the other variables. It is possible, however, that locally suboptimal tests are required in order to find the globally optimal tree.

In our toy examples, the size of the tree did not matter since we retained all of nodes. In a real application, however, the tree's growth could be limited by pruning or similar mechanisms. Pruning trees with different shapes can produce trees with different performances. In practice, locally optimal decisions that are guided by the information gain or Gini impurity heuristics often result in an acceptable decision trees.

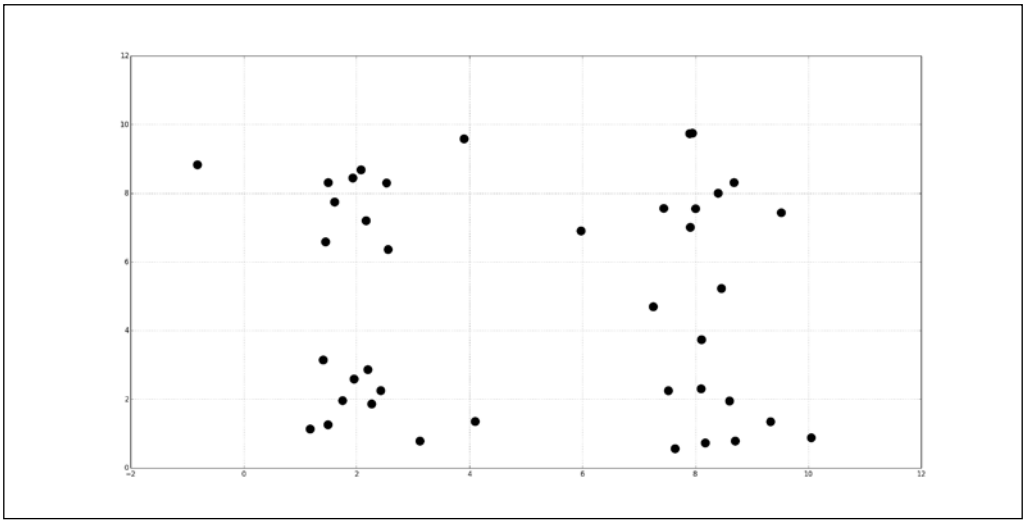
# Summary

In this chapter we learned about simple nonlinear models for classification and regression called decision trees. Like the parlor game Twenty Questions, decision trees are composed of sequences of questions that examine a test instance. The branches of a decision tree terminate in leaves that specify the predicted value of the response variable. We discussed how to train decision trees using the ID3 algorithm, which recursively splits the training instances into subsets that reduce our uncertainty about the value of the response variable. We also discussed ensemble learning methods, which combine the predictions from a set of models to produce an estimator with better predictive performance. Finally, we used random forests to predict whether or not an image on a web page is a banner advertisement. In the next chapter, we will introduce our first unsupervised learning task: clustering.

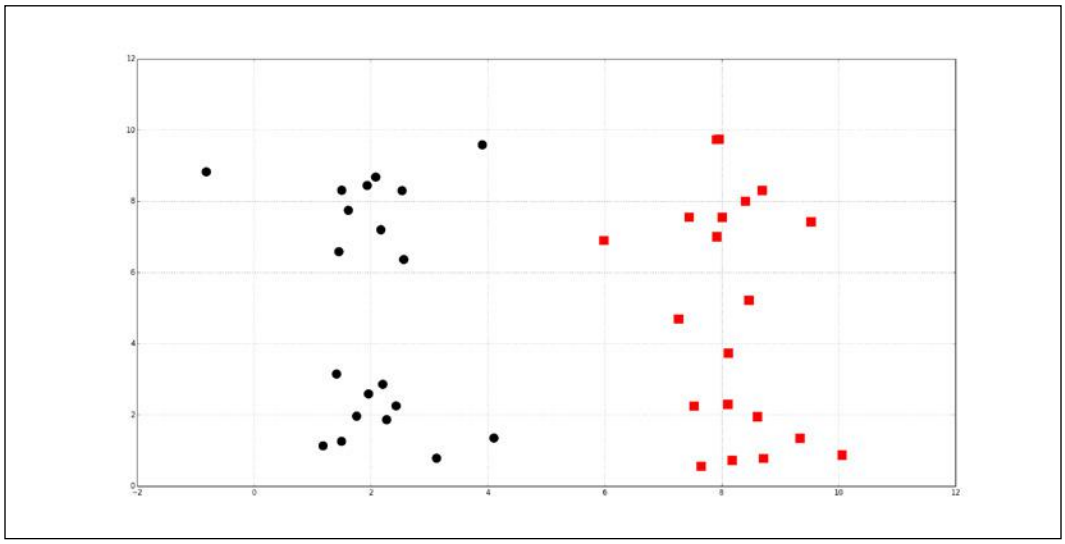
# 6
# Clustering with K-Means

In the previous chapters we discussed supervised learning tasks; we examined algorithms for regression and classification that learned from labeled training data. In this chapter we will discuss an unsupervised learning task called clustering. Clustering is used to find groups of similar observations within a set of unlabeled data. We will discuss the K-Means clustering algorithm, apply it to an image compression problem, and learn to measure its performance. Finally, we will work through a semi-supervised learning problem that combines clustering with classification.

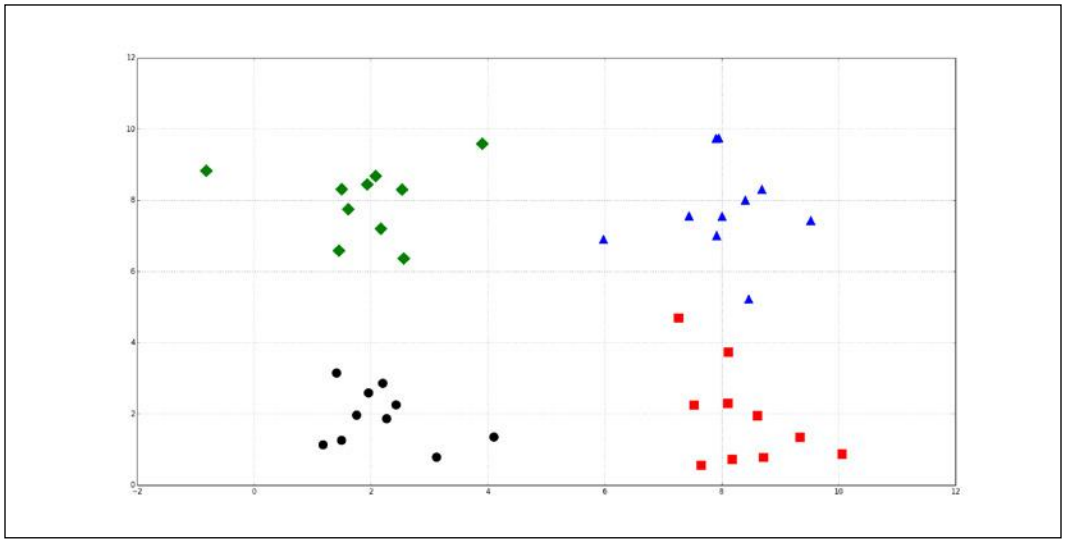
Recall from *Chapter 1*, *The Fundamentals of Machine Learning*, that the goal of unsupervised learning is to discover hidden structure or patterns in unlabeled training data. **Clustering**, or **cluster analysis**, is the task of grouping observations such that members of the same group, or cluster, are more similar to each other by a given metric than they are to the members of the other clusters. As with supervised learning, we will represent an observation as an *n*-dimensional vector. For example, assume that your training data consists of the samples plotted in the following figure:



Clustering might reveal the following two groups, indicated by squares and circles:



Clustering could also reveal the following four groups:



Clustering is commonly used to explore a dataset. Social networks can be clustered to identify communities and to suggest missing connections between people. In biology, clustering is used to find groups of genes with similar expression patterns. Recommendation systems sometimes employ clustering to identify products or media that might appeal to a user. In marketing, clustering is used to find segments of similar consumers. In the following sections, we will work through an example of using the K-Means algorithm to cluster a dataset.

# Clustering with the K-Means algorithm

The K-Means algorithm is a clustering method that is popular because of its speed and scalability. K-Means is an iterative process of moving the centers of the clusters, or the **centroids**, to the mean position of their constituent points, and re-assigning instances to their closest clusters. The titular $K$ is a hyperparameter that specifies the number of clusters that should be created; K-Means automatically assigns observations to clusters but cannot determine the appropriate number of clusters. $K$ must be a positive integer that is less than the number of instances in the training set. Sometimes, the number of clusters is specified by the clustering problem's context. For example, a company that manufactures shoes might know that it is able to support manufacturing three new models. To understand what groups of customers to target with each model, it surveys customers and creates three clusters from the results. That is, the value of $K$ was specified by the problem's context. Other problems may not require a specific number of clusters, and the optimal number of clusters may be ambiguous. We will discuss a heuristic to estimate the optimal number of clusters called the elbow method later in this chapter.

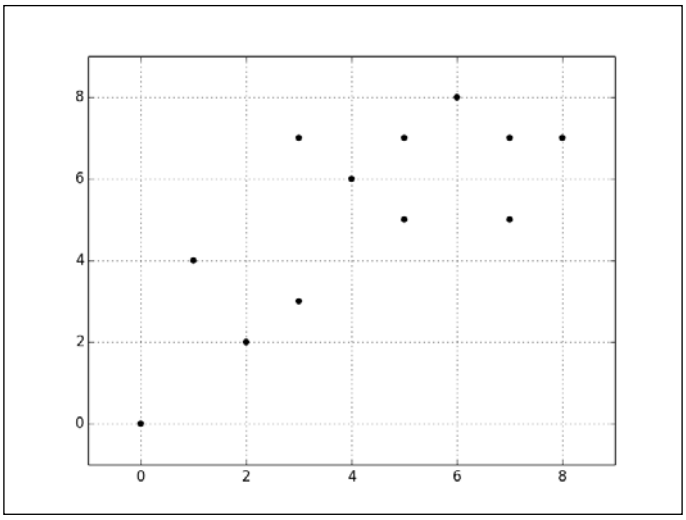
The parameters of K-Means are the positions of the clusters' centroids and the observations that are assigned to each cluster. Like generalized linear models and decision trees, the optimal values of K-Means' parameters are found by minimizing a cost function. The cost function for K-Means is given by the following equation:

$$J = \sum_{k=1}^{K} \sum_{i \in C_k} \left\| x_i - \mu_k \right\|^2$$

In the preceding equation, $\mu_k$ is the centroid for the cluster $k$. The cost function sums the distortions of the clusters. Each cluster's distortion is equal to the sum of the squared distances between its centroid and its constituent instances. The distortion is small for compact clusters and large for clusters that contain scattered instances. The parameters that minimize the cost function are learned through an iterative process of assigning observations to clusters and then moving the clusters. First, the clusters' centroids are initialized to random positions. In practice, setting the centroids' positions equal to the positions of randomly selected observations yields the best results. During each iteration, K-Means assigns observations to the cluster that they are closest to, and then moves the centroids to their assigned observations' mean location. Let's work through an example by hand using the training data shown in the following table:

| **Instance** | **X0** | **X1** |
|---|---|---|
| 1 | 7 | 5 |
| 2 | 5 | 7 |
| 3 | 7 | 7 |
| 4 | 3 | 3 |
| 5 | 4 | 6 |
| 6 | 1 | 4 |
| 7 | 0 | 0 |
| 8 | 2 | 2 |
| 9 | 8 | 7 |
| 10 | 6 | 8 |
| 11 | 5 | 5 |
| 12 | 3 | 7 |

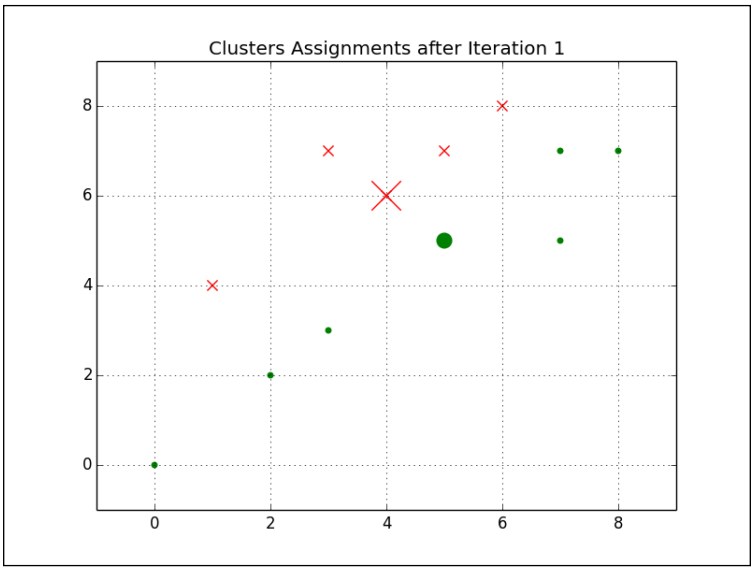
There are two explanatory variables and each instance has two features. The instances are plotted in the following figure:




Assume that K-Means initializes the centroid for the first cluster to the fifth instance and the centroid for the second cluster to the eleventh instance. For each instance, we will calculate its distance to both centroids, and assign it to the cluster with the closest centroid. The initial assignments are shown in the **Cluster** column of the following table:

| Instance | X0 | X1 | C1 distance | C2 distance | Last cluster | Cluster | Changed? |
|---|---|---|---|---|---|---|---|
| 1 | 7 | 5 | 3.16228 | 2 | None | C2 | Yes |
| 2 | 5 | 7 | 1.41421 | 2 | None | C1 | Yes |
| 3 | 7 | 7 | 3.16228 | 2.82843 | None | C2 | Yes |
| 4 | 3 | 3 | 3.16228 | 2.82843 | None | C2 | Yes |
| 5 | 4 | 6 | 0 | 1.41421 | None | C1 | Yes |
| 6 | 1 | 4 | 3.60555 | 4.12311 | None | C1 | Yes |
| 7 | 0 | 0 | 7.21110 | 7.07107 | None | C2 | Yes |
| 8 | 2 | 2 | 4.47214 | 4.24264 | None | C2 | Yes |
| 9 | 8 | 7 | 4.12311 | 3.60555 | None | C2 | Yes |
| 10 | 6 | 8 | 2.82843 | 3.16228 | None | C1 | Yes |
| 11 | 5 | 5 | 1.41421 | 0 | None | C2 | Yes |
| 12 | 3 | 7 | 1.41421 | 2.82843 | None | C1 | Yes |
| C1 centroid | 4 | 6 | | | | | |
| C2 centroid | 5 | 5 | | | | | |

The plotted centroids and the initial cluster assignments are shown in the following graph. Instances assigned to the first cluster are marked with **Xs**, and instances assigned to the second cluster are marked with dots. The markers for the centroids are larger than the markers for the instances.

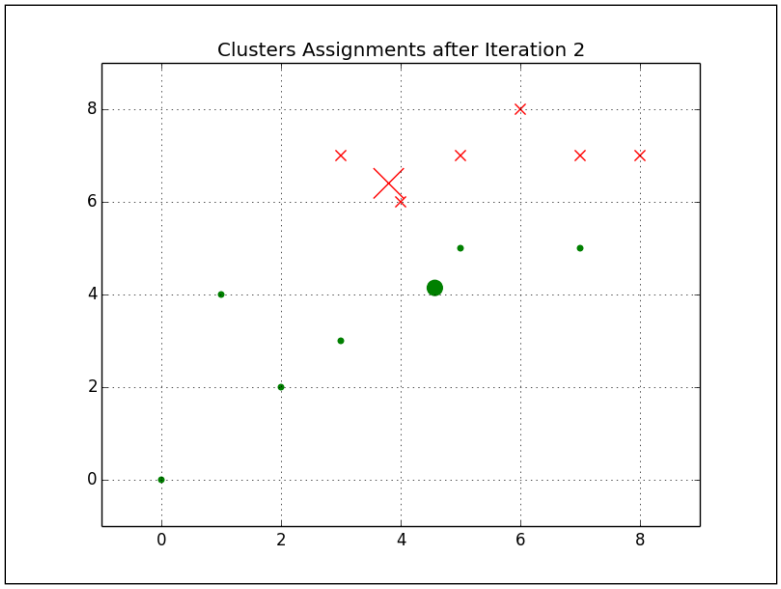



Now we will move both centroids to the means of their constituent instances, recalculate the distances of the training instances to the centroids, and reassign the instances to the closest centroids:

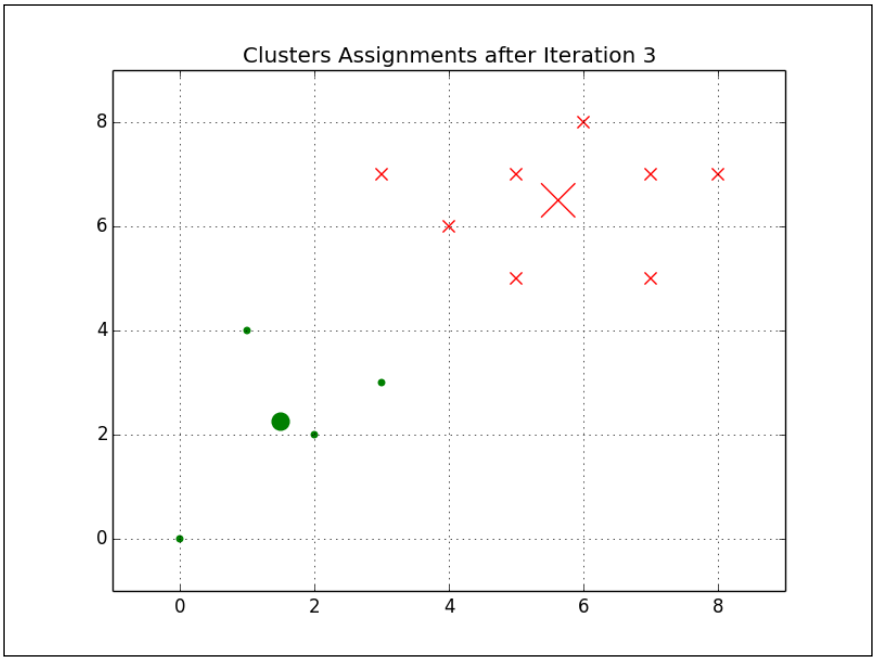
| Instance | X0 | X1 | C1 distance | C2 distance | Last Cluster | New Cluster | Changed? |
|---|---|---|---|---|---|---|---|
| 1 | 7 | 5 | 3.492850 | 2.575394 | C2 | C2 | No |
| 2 | 5 | 7 | 1.341641 | 2.889107 | C1 | C1 | No |
| 3 | 7 | 7 | 3.255764 | 3.749830 | C2 | C1 | Yes |
| 4 | 3 | 3 | 3.492850 | 1.943067 | C2 | C2 | No |
| 5 | 4 | 6 | 0.447214 | 1.943067 | C1 | C1 | No |
| 6 | 1 | 4 | 3.687818 | 3.574285 | C1 | C2 | Yes |
| 7 | 0 | 0 | 7.443118 | 6.169378 | C2 | C2 | No |

| Instance | X0 | X1 | C1 distance | C2 distance | Last Cluster | New Cluster | Changed? |
|---|---|---|---|---|---|---|---|
| 8 | 2 | 2 | 4.753946 | 3.347250 | C2 | C2 | No |
| 9 | 8 | 7 | 4.242641 | 4.463000 | C2 | C1 | Yes |
| 10 | 6 | 8 | 2.720294 | 4.113194 | C1 | C1 | No |
| 11 | 5 | 5 | 1.843909 | 0.958315 | C2 | C2 | No |
| 12 | 3 | 7 | 1 | 3.260775 | C1 | C1 | No |
| C1 centroid | 3.8 | 6.4 | | | | | |
| C2 centroid | 4.571429 | 4.142857 | | | | | |

The new clusters are plotted in the following graph. Note that the centroids are diverging and several instances have changed their assignments:



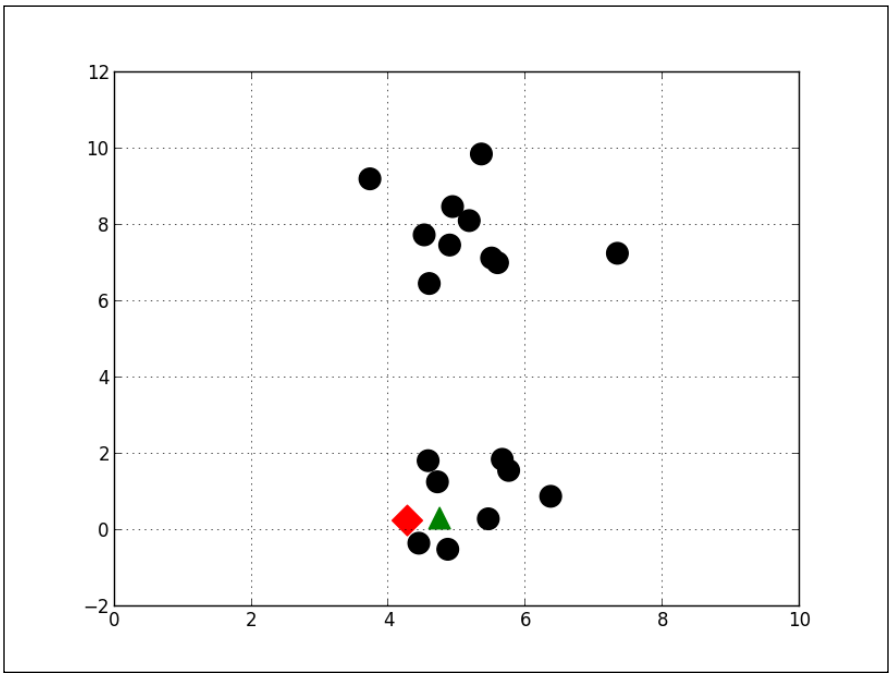
Now, we will move the centroids to the means of their constituents' locations again and reassign the instances to their nearest centroids. The centroids continue to diverge, as shown in the following figure:



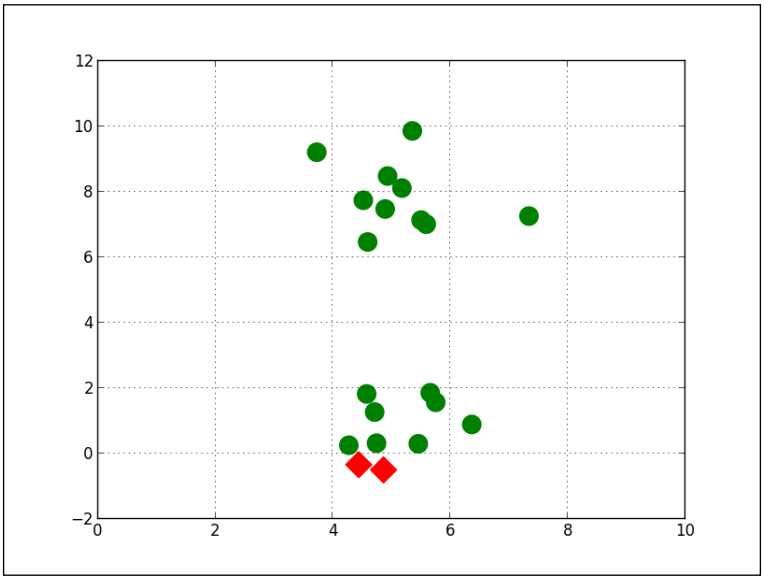

None of the instances' centroid assignments will change in the next iteration; K-Means will continue iterating until some stopping criteria is satisfied. Usually, this criterion is either a threshold for the difference between the values of the cost function for subsequent iterations, or a threshold for the change in the positions of the centroids between subsequent iterations. If these stopping criteria are small enough, K-Means will converge on an optimum. This optimum will not necessarily be the global optimum.

# Local optima

Recall that K-Means initially sets the positions of the clusters' centroids to the positions of randomly selected observations. Sometimes, the random initialization is unlucky and the centroids are set to positions that cause K-Means to converge to a local optimum. For example, assume that K-Means randomly initializes two cluster centroids to the following positions:

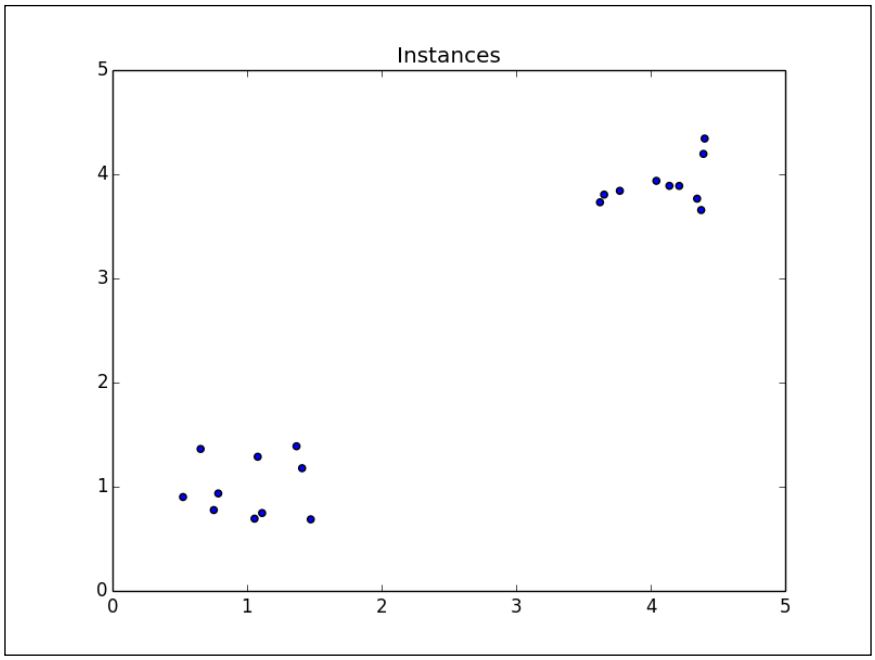


K-Means will eventually converge on a local optimum like that shown in the following figure. These clusters may be informative, but it is more likely that the top and bottom groups of observations are more informative clusters. To avoid local optima, K-Means is often repeated dozens or even hundreds of times. In each iteration, it is randomly initialized to different starting cluster positions. The initialization that minimizes the cost function best is selected.




# The elbow method

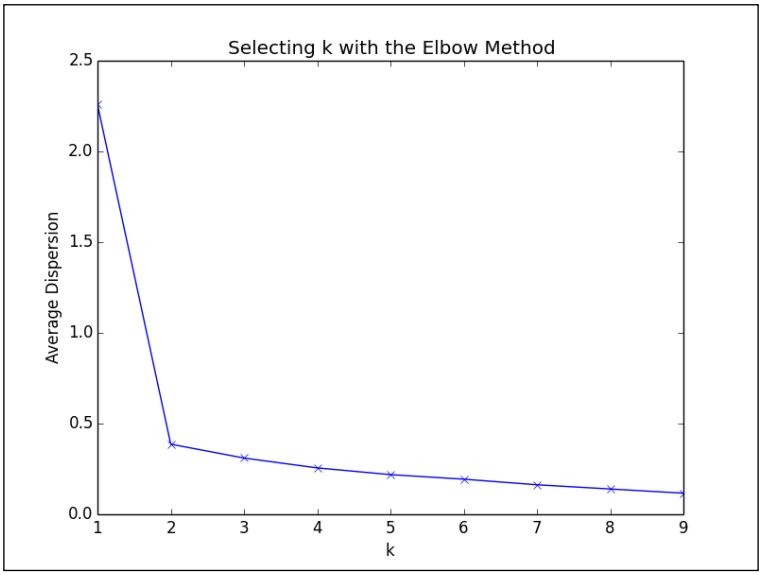
If $K$ is not specified by the problem's context, the optimal number of clusters can be estimated using a technique called the **elbow method**. The elbow method plots the value of the cost function produced by different values of $K$. As $K$ increases, the average distortion will decrease; each cluster will have fewer constituent instances, and the instances will be closer to their respective centroids. However, the improvements to the average distortion will decline as $K$ increases. The value of $K$ at which the improvement to the distortion declines the most is called the elbow. Let's use the elbow method to choose the number of clusters for a dataset. The following scatter plot visualizes a dataset with two obvious clusters:



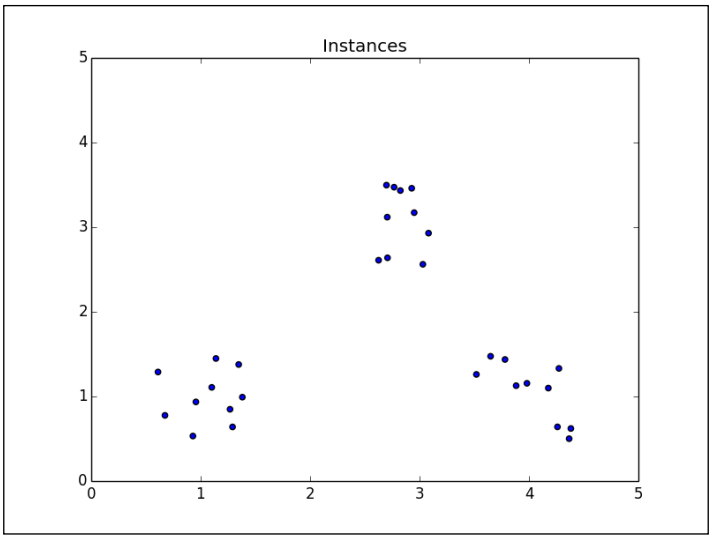

We will calculate and plot the mean distortion of the clusters for each value of $K$ from 1 to 10 with the following code:

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> from scipy.spatial.distance import cdist
>>> import matplotlib.pyplot as plt

>>> cluster1 = np.random.uniform(0.5, 1.5, (2, 10))
>>> cluster2 = np.random.uniform(3.5, 4.5, (2, 10))
>>> X = np.hstack((cluster1, cluster2)).T
>>> X = np.vstack((x, y)).T

>>> K = range(1, 10)
>>> meandistortions = []
>>> for k in K:
>>>     kmeans = KMeans(n_clusters=k)
```

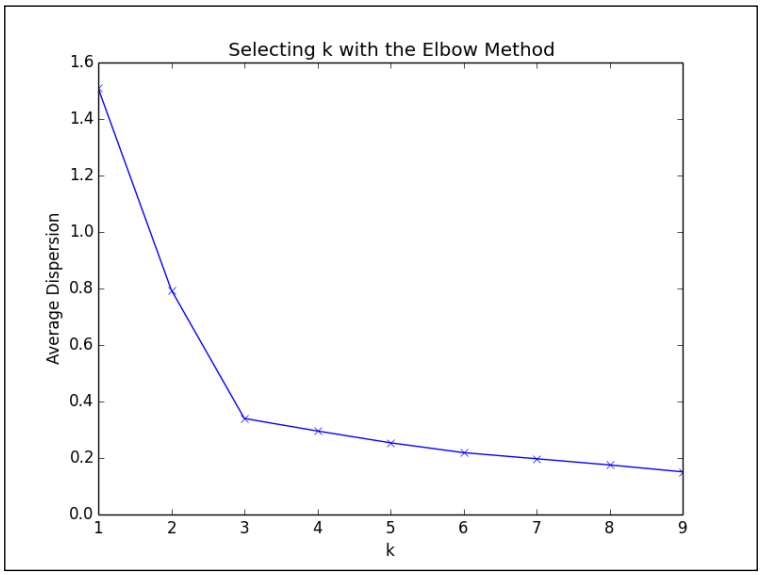
```
>>>     kmeans.fit(X)
>>>     meandistortions.append(sum(np.min(cdist(X, kmeans.cluster_
centers_, 'euclidean'), axis=1)) / X.shape[0])

>>> plt.plot(K, meandistortions, 'bx-')
>>> plt.xlabel('k')
>>> plt.ylabel('Average distortion')
>>> plt.title('Selecting k with the Elbow Method')
>>> plt.show()
```




The average distortion improves rapidly as we increase $K$ from **1** to **2**. There is little improvement for values of $K$ greater than 2. Now let's use the elbow method on the following dataset with three clusters:




The following figure shows the elbow plot for the dataset. From this, we can see that the rate of improvement to the average distortion declines the most when adding a fourth cluster, that is, the elbow method confirms that $K$ should be set to three for this dataset.



# Evaluating clusters

We defined machine learning as the design and study of systems that learn from experience to improve their performance of a task as measured by a given metric. K-Means is an unsupervised learning algorithm; there are no labels or ground truth to compare with the clusters. However, we can still evaluate the performance of the algorithm using intrinsic measures. We have already discussed measuring the distortions of the clusters. In this section, we will discuss another performance measure for clustering called the **silhouette coefficient**. The silhouette coefficient is a measure of the compactness and separation of the clusters. It increases as the quality of the clusters increase; it is large for compact clusters that are far from each other and small for large, overlapping clusters. The silhouette coefficient is calculated per instance; for a set of instances, it is calculated as the mean of the individual samples' scores. The silhouette coefficient for an instance is calculated with the following equation:

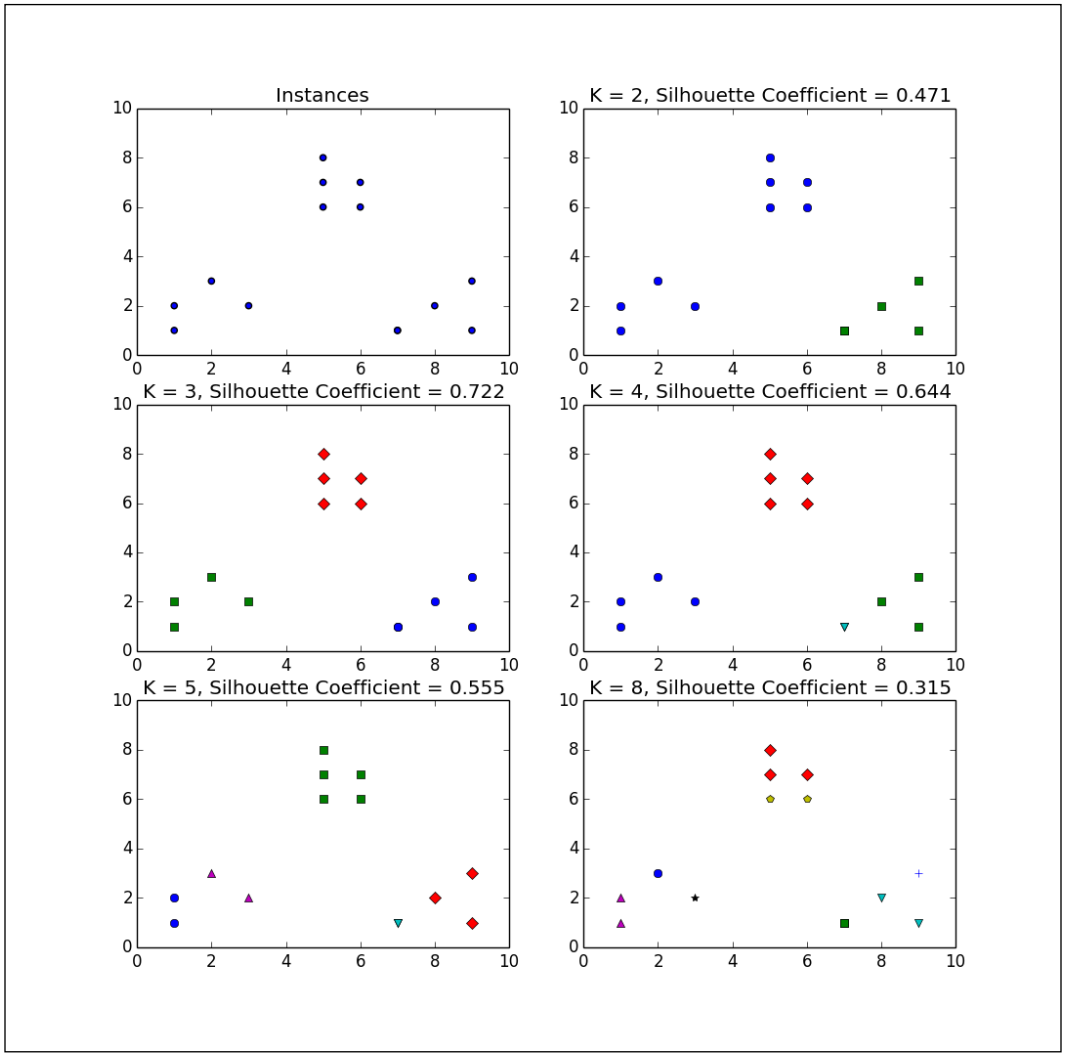
$$s = \frac{ba}{\max(a, b)}$$

$a$ is the mean distance between the instances in the cluster. $b$ is the mean distance between the instance and the instances in the next closest cluster. The following example runs K-Means four times to create two, three, four, and eight clusters from a toy dataset and calculates the silhouette coefficient for each run:

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> from sklearn import metrics
>>> import matplotlib.pyplot as plt
>>> plt.subplot(3, 2, 1)
>>> x1 = np.array([1, 2, 3, 1, 5, 6, 5, 5, 6, 7, 8, 9, 7, 9])
>>> x2 = np.array([1, 3, 2, 2, 8, 6, 7, 6, 7, 1, 2, 1, 1, 3])
>>> X = np.array(zip(x1, x2)).reshape(len(x1), 2)
>>> plt.xlim([0, 10])
>>> plt.ylim([0, 10])
>>> plt.title('Instances')
>>> plt.scatter(x1, x2)
>>> colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'b']
>>> markers = ['o', 's', 'D', 'v', '^', 'p', '*', '+']
>>> tests = [2, 3, 4, 5, 8]
>>> subplot_counter = 1
>>> for t in tests:
>>>     subplot_counter += 1
>>>     plt.subplot(3, 2, subplot_counter)
>>>     kmeans_model = KMeans(n_clusters=t).fit(X)
```

```
>>>     for i, l in enumerate(kmeans_model.labels_):
>>>         plt.plot(x1[i], x2[i], color=colors[l], marker=markers[l],
ls='None')
>>>     plt.xlim([0, 10])
>>>     plt.ylim([0, 10])
>>>     plt.title('K = %s, silhouette coefficient = %.03f' % (
>>>         t, metrics.silhouette_score(X, kmeans_model.labels_,
metric='euclidean')))
>>> plt.show()
```

This script produces the following figure:



The dataset contains three obvious clusters. Accordingly, the silhouette coefficient is greatest when $K$ is equal to three. Setting $K$ equal to eight produces clusters of instances that are as close to each other as they are to the instances in some of the other clusters, and the silhouette coefficient of these clusters is smallest.

# Image quantization

In the previous sections, we used clustering to explore the structure of a dataset. Now let's apply it to a different problem. Image quantization is a lossy compression method that replaces a range of similar colors in an image with a single color. Quantization reduces the size of the image file since fewer bits are required to represent the colors. In the following example, we will use clustering to discover a compressed palette for an image that contains its most important colors. We will then rebuild the image using the compressed palette. This example requires the `mahotas` image processing library, which can be installed using `pip install mahotas`:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.cluster import KMeans
>>> from sklearn.utils import shuffle
>>> import mahotas as mh
```

First we read and flatten the image:

```
>>> original_img = np.array(mh.imread('img/tree.jpg'), dtype=np.
float64) / 255
>>> original_dimensions = tuple(original_img.shape)
>>> width, height, depth = tuple(original_img.shape)
>>> image_flattened = np.reshape(original_img, (width * height, depth))
```

We then use K-Means to create 64 clusters from a sample of 1,000 randomly selected colors. Each of the clusters will be a color in the compressed palette. The code is as follows:

```
>>> image_array_sample = shuffle(image_flattened, random_state=0)
[:1000]
>>> estimator = KMeans(n_clusters=64, random_state=0)
>>> estimator.fit(image_array_sample)
```

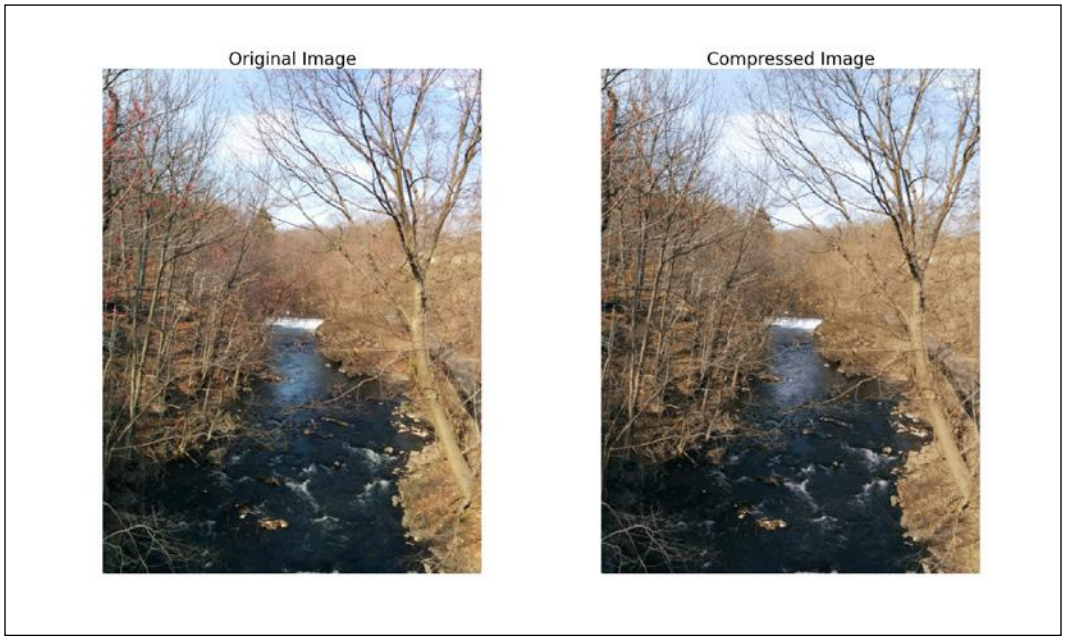
Next, we predict the cluster assignment for each of the pixels in the original image:

```
>>> cluster_assignments = estimator.predict(image_flattened)
```

Finally, we create the compressed image from the compressed palette and cluster assignments:

```
>>> compressed_palette = estimator.cluster_centers_
>>> compressed_img = np.zeros((width, height, compressed_palette.
shape[1]))
>>> label_idx = 0
>>> for i in range(width):
>>>     for j in range(height):
>>>         compressed_img[i][j] = compressed_palette[cluster_
assignments[label_idx]]
>>>         label_idx += 1
>>> plt.subplot(122)
>>> plt.title('Original Image')
>>> plt.imshow(original_img)
>>> plt.axis('off')
>>> plt.subplot(121)
>>> plt.title('Compressed Image')
>>> plt.imshow(compressed_img)
>>> plt.axis('off')
>>> plt.show()
```

The original and compressed versions of the image are show in the following figure:



# Clustering to learn features

In this example, we will combine clustering with classification in a semi-supervised learning problem. You will learn features by clustering unlabeled data and use the learned features to build a supervised classifier.

Suppose you own a cat and a dog. Suppose that you have purchased a smartphone, ostensibly to use to communicate with humans, but in practice just to use to photograph your cat and dog. Your photographs are awesome and you are certain that your friends and co-workers would love to review all of them in detail. You'd like to be courteous and respect that some people will only want to see your cat photos, while others will only want to see your dog photos, but separating the photos is laborious. Let's build a semi-supervised learning system that can classify images of cats and dogs.

Recall from *Chapter 3*, *Feature Extraction and Preprocessing*, that a naïve approach to classifying images is to use the intensities, or brightnesses, of all of the pixels as explanatory variables. This approach produces high-dimensional feature vectors for even small images. Unlike the high-dimensional feature vectors we used to represent documents, these vectors are not sparse. Furthermore, it is obvious that this approach is sensitive to the image's illumination, scale, and orientation. In *Chapter 3*, *Feature Extraction and Preprocessing*, we also discussed SIFT and SURF descriptors, which describe interesting regions of an image in ways that are invariant to scale, rotation, and illumination. In this example, we will cluster the descriptors extracted from all of the images to learn features. We will then represent an image with a vector with one element for each cluster. Each element will encode the number of descriptors extracted from the image that were assigned to the cluster. This approach is sometimes called the **bag-of-features** representation, as the collection of clusters is analogous to the bag-of-words representation's vocabulary. We will use 1,000 images of cats and 1,000 images of dogs from the training set for Kaggle's *Dogs vs. Cats* competition. The dataset can be downloaded from `https://www.kaggle.com/c/dogs-vs-cats/data`. We will label cats as the positive class and dogs as the negative class. Note that the images have different dimensions; since our feature vectors do not represent pixels, we do not need to resize the images to have the same dimensions. We will train using the first 60 percent of the images, and test on the remaining 40 percent:

```
>>> import numpy as np
>>> import mahotas as mh
>>> from mahotas.features import surf
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.metrics import *
>>> from sklearn.cluster import MiniBatchKMeans
>>> import glob
```

First, we load the images, convert them to grayscale, and extract the SURF descriptors. SURF descriptors can be extracted more quickly than many similar features, but extracting descriptors from 2,000 images is still computationally expensive. Unlike the previous examples, this script requires several minutes to execute on most computers:

```
>>> all_instance_filenames = []
>>> all_instance_targets = []
>>> for f in glob.glob('cats-and-dogs-img/*.jpg'):
>>>     target = 1 if 'cat' in f else 0
>>>     all_instance_filenames.append(f)
>>>     all_instance_targets.append(target)
>>> surf_features = []
>>> counter = 0
>>> for f in all_instance_filenames:
>>>     print 'Reading image:', f
>>>     image = mh.imread(f, as_grey=True)
>>>     surf_features.append(surf.surf(image)[:, 5:])

>>> train_len = int(len(all_instance_filenames) * .60)
>>> X_train_surf_features = np.concatenate(surf_features[:train_len])
>>> X_test_surf_feautres = np.concatenate(surf_features[train_len:])
>>> y_train = all_instance_targets[:train_len]
>>> y_test = all_instance_targets[train_len:]
```

We then group the extracted descriptors into 300 clusters in the following code sample. We use `MiniBatchKMeans`, a variation of K-Means that uses a random sample of the instances in each iteration. As it computes the distances to the centroids for only a sample of the instances in each iteration, `MiniBatchKMeans` converges more quickly but its clusters' distortions may be greater. In practice, the results are similar, and this compromise is acceptable.:

```
>>> n_clusters = 300
>>> print 'Clustering', len(X_train_surf_features), 'features'
>>> estimator = MiniBatchKMeans(n_clusters=n_clusters)
>>> estimator.fit_transform(X_train_surf_features)
```

Next, we construct feature vectors for the training and testing data. We find the cluster associated with each of the extracted SURF descriptors, and count them using NumPy's `binCount()` function. The following code produces a 300-dimensional feature vector for each instance:

```
>>> X_train = []
>>> for instance in surf_features[:train_len]:
>>>     clusters = estimator.predict(instance)
>>>     features = np.bincount(clusters)
>>>     if len(features) < n_clusters:
>>>         features = np.append(features, np.zeros((1, n_clusters-
len(features))))
>>>     X_train.append(features)

>>> X_test = []
>>> for instance in surf_features[train_len:]:
>>>     clusters = estimator.predict(instance)
>>>     features = np.bincount(clusters)
>>>     if len(features) < n_clusters:
>>>         features = np.append(features, np.zeros((1, n_clusters-
len(features))))
>>>     X_test.append(features)
```

Finally, we train a logistic regression classifier on the feature vectors and targets, and assess its precision, recall, and accuracy:

```
>>> clf = LogisticRegression(C=0.001, penalty='l2')
>>> clf.fit_transform(X_train, y_train)
>>> predictions = clf.predict(X_test)
>>> print classification_report(y_test, predictions)
>>> print 'Precision: ', precision_score(y_test, predictions)
>>> print 'Recall: ', recall_score(y_test, predictions)
>>> print 'Accuracy: ', accuracy_score(y_test, predictions)

Reading image: dog.9344.jpg
...
Reading image: dog.8892.jpg
```

```
Clustering 756914 features
             precision    recall  f1-score   support

          0       0.71      0.76      0.73       392
          1       0.75      0.70      0.72       408

avg / total       0.73      0.73      0.73       800

Precision:  0.751322751323
Recall:  0.696078431373
Accuracy:  0.7275
```

This semi-supervised system has better precision and recall than a logistic regression classifier that uses only the pixel intensities as features. Furthermore, our feature representations have only 300 dimensions; even small 100 x 100 pixel images would have 10,000 dimensions.

# Summary

In this chapter, we discussed our first unsupervised learning task: clustering. Clustering is used to discover structure in unlabeled data. You learned about the K-Means clustering algorithm, which iteratively assigns instances to clusters and refines the positions of the cluster centroids. While K-Means learns from experience without supervision, its performance is still measurable; you learned to use distortion and the silhouette coefficient to evaluate clusters. We applied K-Means to two different problems. First, we used K-Means for image quantization, a compression technique that represents a range of colors with a single color. We also used K-Means to learn features in a semi-supervised image classification problem.

In the next chapter, we will discuss another unsupervised learning task called dimensionality reduction. Like the semi-supervised feature representations we created to classify images of cats and dogs, dimensionality reduction can be used to reduce the dimensions of a set of explanatory variables while retaining as much information as possible.

# 7
# Dimensionality Reduction with PCA

In this chapter, we will discuss a technique for reducing the dimensions of data called **Principal Component Analysis** (**PCA**). Dimensionality reduction is motivated by several problems. First, it can be used to mitigate problems caused by the curse of dimensionality. Second, dimensionality reduction can be used to compress data while minimizing the amount of information that is lost. Third, understanding the structure of data with hundreds of dimensions can be difficult; data with only two or three dimensions can be visualized easily. We will use PCA to visualize a high-dimensional dataset in two dimensions, and build a face recognition system.

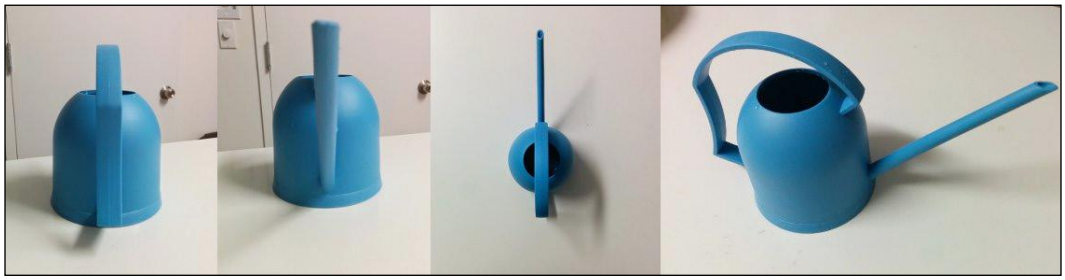
## An overview of PCA

Recall from *Chapter 3*, *Feature Extraction and Preprocessing*, that problems involving high-dimensional data can be affected by the curse of dimensionality. As the dimensions of a data set increases, the number of samples required for an estimator to generalize increases exponentially. Acquiring such large data may be infeasible in some applications, and learning from large data sets requires more memory and processing power. Furthermore, the sparseness of data often increases with its dimensions. It can become more difficult to detect similar instances in high-dimensional space as all of the instances are similarly sparse.

Principal Component Analysis, also known as the Karhunen-Loeve Transform, is a technique used to search for patterns in high-dimensional data. PCA is commonly used to explore and visualize high-dimensional data sets. It can also be used to compress data, and process data before it is used by another estimator. PCA reduces a set of possibly-correlated, high-dimensional variables to a lower-dimensional set of linearly uncorrelated synthetic variables called **principal components**. The lower-dimensional data will preserve as much of the variance of the original data as possible.

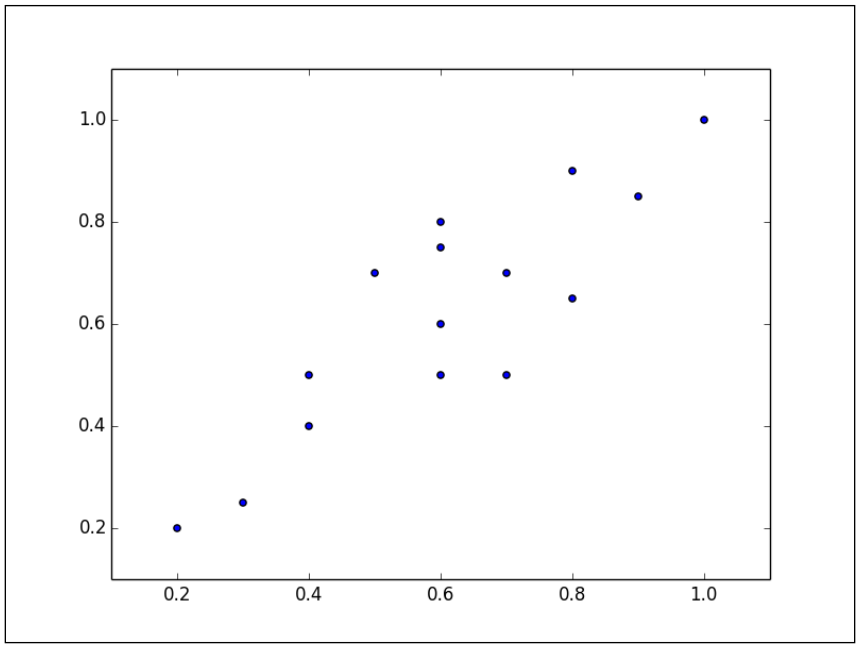
PCA reduces the dimensions of a data set by projecting the data onto a lower-dimensional subspace. For example, a two dimensional data set could be reduced by projecting the points onto a line; each instance in the data set would then be represented by a single value rather than a pair of values. A three-dimensional dataset could be reduced to two dimensions by projecting the variables onto a plane. In general, an $n$-dimensional dataset can be reduced by projecting the dataset onto a $k$-dimensional subspace, where $k$ is less than $n$. More formally, PCA can be used to find a set of vectors that span a subspace, which minimizes the sum of the squared errors of the projected data. This projection will retain the greatest proportion of the original data set's variance.

Imagine that you are a photographer for a gardening supply catalog, and that you are tasked with photographing a watering can. The watering can is three-dimensional, but the photograph is two-dimensional; you must create a two-dimensional representation that describes as much of the watering can as possible. The following are four possible pictures that you could use:

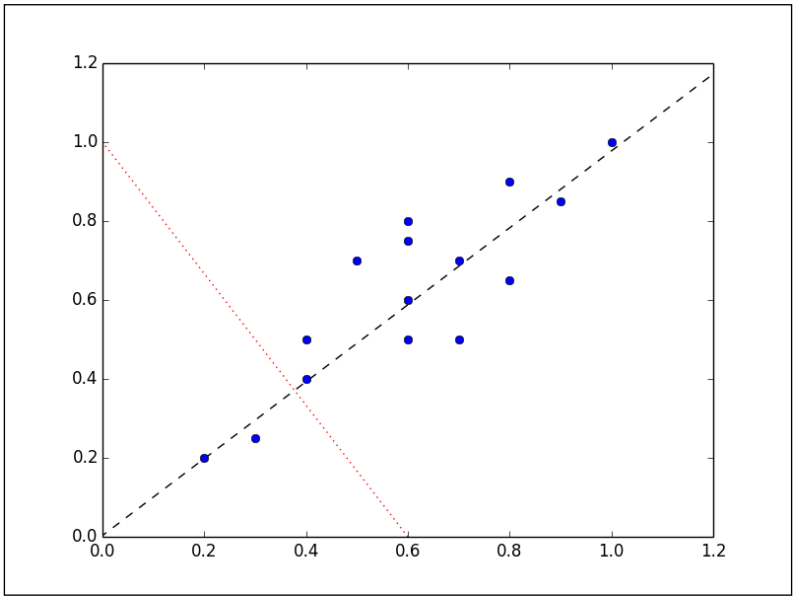


In the first photograph, the back of the watering can is visible, but the front cannot be seen. The second picture is angled to look directly down the spout of the watering can; this picture provides information about the front of the can that was not visible in the first photograph, but now the handle cannot be seen. The height of the watering can cannot be discerned from the bird's eye view of the third picture. The fourth picture is the obvious choice for the catalog; the watering can's height, top, spout, and handle are all discernible in this image.

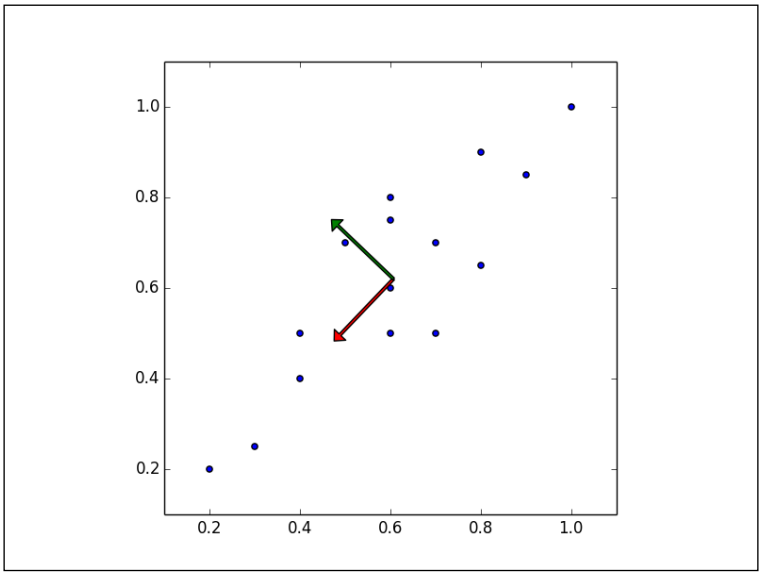
The motivation of PCA is similar; it can project data in a high-dimensional space to a lower-dimensional space that retains as much of the variance as possible. PCA rotates the data set to align with its principal components to maximize the variance contained within the first several principal components. Assume that we have the data set that is plotted in the following figure:




The instances approximately form a long, thin ellipse stretching from the origin to the top right of the plot. To reduce the dimensions of this data set, we must project the points onto a line. The following are two lines that the data could be projected onto. Along which line do the instances vary the most?

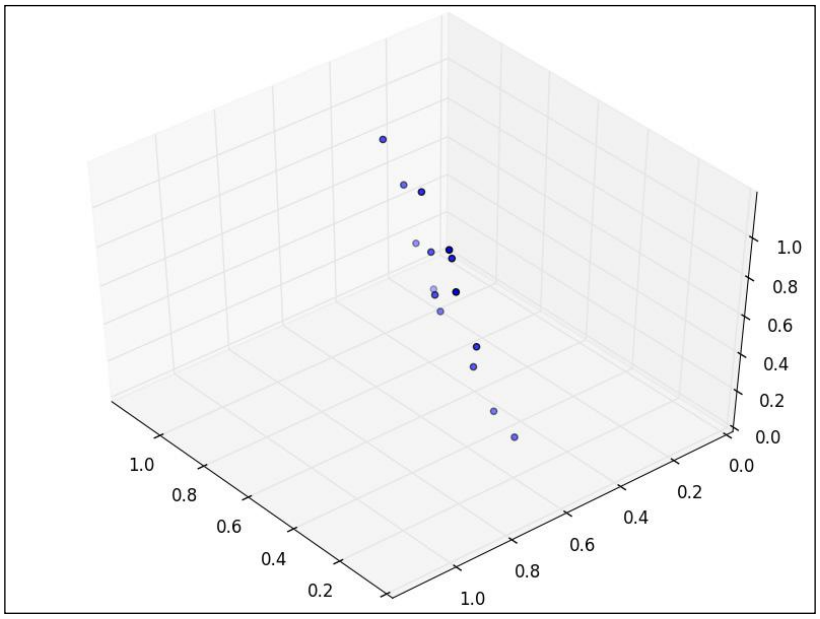


The instances vary more along the dashed line than the dotted line. In fact, the dashed line is the first principal component. The second principal component must be orthogonal to the first principal component; that is, the second principal component must be statistically independent, and will appear to be perpendicular to the first principal component when it is plotted, as shown in the following figure:




Each subsequent principal component preserves the maximum amount of the remaining variance; the only constraint is that each must be orthogonal to the other principal components.

Now assume that the data set is three dimensional. The scatter plot of the points looks like a flat disc that has been rotated slightly about one of the axes.




The points can be rotated and translated such that the tilted disk lies almost exactly in two dimensions. The points now form an ellipse; the third dimension contains almost no variance and can be discarded.

PCA is most useful when the variance in a data set is distributed unevenly across the dimensions. Consider a three-dimensional data set with a spherical convex hull. PCA cannot be used effectively with this data set because there is equal variance in each dimension; none of the dimensions can be discarded without losing a significant amount of information.

It is easy to visually identify the principal components of data sets with only two or three dimensions. In the next section, we will discuss how to calculate the principal components of high-dimensional data.

# Performing Principal Component Analysis

There are several terms that we must define before discussing how principal component analysis works.

# Variance, Covariance, and Covariance Matrices

Recall that **variance** is a measure of how a set of values are spread out. Variance is calculated as the average of the squared differences of the values and mean of the values, as per the following equation:

$$s^2 = \frac{\sum_{i=1}^{n}\left(X_i - \bar{X}\right)^2}{n-1}$$

**Covariance** is a measure of how much two variables change together; it is a measure of the strength of the correlation between two sets of variables. If the covariance of two variables is zero, the variables are uncorrelated. Note that uncorrelated variables are not necessarily independent, as correlation is only a measure of linear dependence. The covariance of two variables is calculated using the following equation:

$$\operatorname{cov}\left(X,Y\right) = \frac{\sum_{i=1}^{n}\left(X_i - \bar{X}\right)\left(Y_i - \bar{y}\right)}{n-1}$$

If the covariance is nonzero, the sign indicates whether the variables are positively or negatively correlated. When two variables are positively correlated, one increases as the other increases. When variables are negatively correlated, one variable decreases relative to its mean as the other variable increases relative to its mean. A **covariance matrix** describes the covariance values between each pair of dimensions in a data set. The element $\left(i, j\right)$ indicates the covariance of the $ith$ and $jth$ dimensions of the data. For example, a covariance matrix for a three-dimensional data is given by the following matrix:

$$C = \begin{bmatrix} \operatorname{cov}\left(x_1,x_1\right) & \operatorname{cov}\left(x_1,x_2\right) & \operatorname{cov}\left(x_1,x_3\right) \\ \operatorname{cov}\left(x_2,x_1\right) & \operatorname{cov}\left(x_2,x_2\right) & \operatorname{cov}\left(x_2,x_3\right) \\ \operatorname{cov}\left(x_3,x_1\right) & \operatorname{cov}\left(x_3,x_2\right) & \operatorname{cov}\left(x_3,x_3\right) \end{bmatrix}$$

Let's calculate the covariance matrix for the following data set:

| | | |
|---|---|---|
| 2 | 0 | −1.4 |
| 2.2 | 0.2 | −1.5 |
| 2.4 | 0.1 | −1 |
| 1.9 | 0 | −1.2 |

The means of the variables are 2.125, 0.075, and -1.275. We can then calculate the covariances of each pair of variables to produce the following covariance matrix:

$$C = \begin{bmatrix} 0.0491666667 & 0.0141666667 & 0.0191666667 \\ 0.0141666667 & 0.00916667 & -0.00583333 \\ 0.0191666667 & -0.00583333 & 0.04916667 \end{bmatrix}$$

We can verify our calculations using NumPy:

```
>>> import numpy as np
>>> X = [[2, 0, -1.4],
>>>      [2.2, 0.2, -1.5],
>>>      [2.4, 0.1, -1],
>>>      [1.9, 0, -1.2]]
>>> print np.cov(np.array(X).T)
[[ 0.04916667  0.01416667  0.01916667]
 [ 0.01416667  0.00916667 -0.00583333]
 [ 0.01916667 -0.00583333  0.04916667]]
```

# Eigenvectors and eigenvalues

A vector is described by a **direction** and **magnitude**, or length. An **eigenvector** of a matrix is a non-zero vector that satisfies the following equation:

$$A\vec{v} = \lambda\vec{v}$$

In the preceding equation, $\vec{v}$ is an eigenvector, $A$ is a square matrix, and $\lambda$ is a scalar called an **eigenvalue**. The direction of an eigenvector remains the same after it has been transformed by $A$; only its magnitude has changed, as indicated by the eigenvalue; that is, multiplying a matrix by one of its eigenvectors is equal to scaling the eigenvector. The prefix *eigen* is the German word for *belonging to* or *peculiar to*; the eigenvectors of a matrix are the vectors that *belong* to and characterize the structure of the data.

Eigenvectors and eigenvalues can only be derived from square matrices, and not all square matrices have eigenvectors or eigenvalues. If a matrix does have eigenvectors and eigenvalues, it will have a pair for each of its dimensions. The principal components of a matrix are the eigenvectors of its covariance matrix, ordered by their corresponding eigenvalues. The eigenvector with the greatest eigenvalue is the first principal component; the second principal component is the eigenvector with the second greatest eigenvalue, and so on.

Let's calculate the eigenvectors and eigenvalues of the following matrix:

$$\mathrm{A} = \begin{bmatrix} 1 & -2 \\ 2 & -3 \end{bmatrix}$$

Recall that the product of *A* and any eigenvector of *A* must be equal to the eigenvector multiplied by its eigenvalue. We will begin by finding the eigenvalues, which we can find using the following characteristic equations:

$$\left(A - \lambda I\right)\vec{v} = 0$$

$$\left|\mathrm{A} - \lambda * \mathrm{I}\right| = \left\| \begin{bmatrix} 1 & -2 \\ 2 & -3 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \right\| = 0$$

The characteristic equation states that the determinant of the matrix, that is, the difference between the data matrix and the product of the identity matrix and an eigenvalue is zero:

$$\left\| \begin{bmatrix} 1-\lambda & -2 \\ 2 & -3-\lambda \end{bmatrix} \right\| = \left(\lambda + 1\right)\left(\lambda + 1\right) = 0$$

Both of the eigenvalues for this matrix are equal to **-1**. We can now use the eigenvalues to solve the eigenvectors:

$$A\vec{v} = \lambda\vec{v}$$

First, we set the equation equal to zero:

$$\left(A - \lambda I\right)\vec{v} = 0$$

Substituting our values for $A$ produces the following:

$$\left(\begin{bmatrix}1 & -2\\ 2 & -3\end{bmatrix}-\begin{bmatrix}\lambda & 0\\ 0 & \lambda\end{bmatrix}\right)\vec{v}=\begin{bmatrix}1-\lambda & -2\\ 2 & -3-\lambda\end{bmatrix}\vec{v}=\begin{bmatrix}1-\lambda & -2\\ 2 & -3-\lambda\end{bmatrix}\begin{bmatrix}v_{1,1}\\ v_{1,2}\end{bmatrix}=0$$

We can then substitute the first eigenvalue in our first eigenvalue to solve the eigenvectors.

$$\begin{bmatrix}1-(-1) & -2\\ 2 & -3-(-1)\end{bmatrix}\begin{bmatrix}v_{1,1}\\ v_{1,2}\end{bmatrix}=\begin{bmatrix}2 & -2\\ 2 & -2\end{bmatrix}\begin{bmatrix}v_{1,1}\\ v_{1,2}\end{bmatrix}=0$$

The preceding equation can be rewritten as a system of equations:

$$\begin{Bmatrix}2v_{1,1}+-\left(2v_{1,2}\right)=0\\ 2v_{1,1}+-\left(2v_{1,2}\right)=0\end{Bmatrix}$$

Any non-zero vector that satisfies the preceding equations, such as the following, can be used as an eigenvector:

$$\begin{bmatrix}1 & -2\\ 2 & -3\end{bmatrix}\begin{bmatrix}1\\ 1\end{bmatrix}=-1\begin{bmatrix}1\\ 1\end{bmatrix}=\begin{bmatrix}-1\\ -1\end{bmatrix}$$

PCA requires unit eigenvectors, or eigenvectors that have a length equal to **1**. We can normalize an eigenvector by dividing it by its norm, which is given by the following equation:

$$\|x\|=\sqrt{x_1^2+x_2^2+\cdots+x_n^2}$$

The norm of our vector is equal to the following:

$$\left\|\begin{bmatrix}1\\ 1\end{bmatrix}\right\|=\sqrt{1^2+1^2}=\sqrt{2}$$

This produces the following unit eigenvector:

$$\begin{bmatrix}1\\1\end{bmatrix} / \sqrt{2} = \begin{bmatrix}0.7071067811865475\\0.7071067811865475\end{bmatrix}$$

We can verify that our solutions for the eigenvectors are correct using NumPy. The `eig` function returns a tuple of the eigenvalues and eigenvectors:

```
>>> import numpy as np
>>> w, v = np.linalg.eig(np.array([[1, -2], [2, -3]]))
>>> w; v
array([-0.99999998, -1.00000002])
array([[ 0.70710678,  0.70710678],
```

# Dimensionality reduction with Principal Component Analysis

Let's use principal component analysis to reduce the following two-dimensional data set to one dimension:

| x1 | x2 |
|---|---|
| 0.9 | 1 |
| 2.4 | 2.6 |
| 1.2 | 1.7 |
| 0.5 | 0.7 |
| 0.3 | 0.7 |
| 1.8 | 1.4 |
| 0.5 | 0.6 |
| 0.3 | 0.6 |
| 2.5 | 2.6 |
| 1.3 | 1.1 |

The first step of PCA is to subtract the mean of each explanatory variable from each observation:

| **x1** | **x2** |
|---|---|
| 0.9 - 1.17 = -0.27 | 1 - 1.3 = -0.3 |
| 2.4 - 1.17 = 1.23 | 2.6 - 1.3 = 1.3 |
| 1.2 - 1.17 = 0.03 | 1.7 - 1.3 = 0.4 |
| 0.5 - 1.17 = -0.67 | -0.7 - 1.3 = 0.6 |
| 0.3 - 1.17 = -0.87 | -0.7 - 1.3 = 0.6 |
| 1.8 - 1.17 = 0.63 | 1.4 - 1.3 = 0.1 |
| 0.5 - 1.17 = -0.67 | 0.6 - 1.3 = -0.7 |
| 0.3 - 1.17 = -0.87 | 0.6 - 1.3 = -0.7 |
| 2.5 - 1.17 = 1.33 | 2.6 - 1.3 = 1.3 |
| 1.3 - 1.17 = 0.13 | 1.1 - 1.3 = -0.2 |

Next, we must calculate the principal components of the data. Recall that the principal components are the eigenvectors of the data's covariance matrix ordered by their eigenvalues. The principal components can be found using two different techniques. The first technique requires calculating the covariance matrix of the data. Since the covariance matrix will be square, we can calculate the eigenvectors and eigenvalues using the approach described in the previous section. The second technique uses singular value decomposition of the data matrix to find the eigenvectors and square roots of the eigenvalues of the covariance matrix. We will work through an example using the first technique, and then describe the second technique that is used by scikit-learn's implementation of PCA.

The following matrix is the covariance matrix for the data:

$$C = \begin{bmatrix} 0.6867777778 & 0.6066666667 \\ 0.6066666667 & 0.5977777778 \end{bmatrix}$$

Using the technique described in the previous section, the eigenvalues are 1.250 and 0.034. The following are the unit eigenvectors:

$$\begin{bmatrix} 0.73251454 & -0.68075138 \\ 0.68075138 & 0.73251454 \end{bmatrix}$$

Next, we will project the data onto the principal components. The first eigenvector has the greatest eigenvalue and is the first principal component. We will build a transformation matrix in which each column of the matrix is the eigenvector for a principal component. If we were reducing a five-dimensional data set to three dimensions, we would build a matrix with three columns. In this example, we will project our two-dimensional data set onto one dimension, so we will use only the eigenvector for the first principal component. Finally, we will find the dot product of the data matrix and transformation matrix. The following is the result of projecting our data onto the first principal component:

$$\begin{bmatrix} -0.27 & -0.3 \\ 1.23 & 1.3 \\ 0.03 & 0.4 \\ -0.67 & -0.6 \\ -0.87 & -0.6 \\ 0.63 & 0.1 \\ -0.67 & -0.7 \\ -0.87 & -0.7 \\ 1.33 & 1.3 \\ 0.13 & -0.2 \end{bmatrix} \begin{bmatrix} 0.73251454 \\ 0.68075138 \end{bmatrix} = \begin{bmatrix} -0.40200434 \\ 1.78596968 \\ 0.29427599 \\ -0.89923557 \\ -1.04573848 \\ 0.5295593 \\ -0.96731071 \\ -1.11381362 \\ 1.85922113 \\ -0.04092339 \end{bmatrix}$$

Many implementations of PCA, including the one of scikit-learn, use singular value decomposition to calculate the eigenvectors and eigenvalues. SVD is given by the following equation:

$$X = U \sum V^T$$

The columns of $U$ are called left singular vectors of the data matrix, the columns of $V$ are its right singular vectors, and the diagonal entries of $\Sigma$ are its singular values. While the singular vectors and values of a matrix are useful in some applications of signal processing and statistics, we are only interested in them as they relate to the eigenvectors and eigenvalues of the data matrix. Specifically, the left singular vectors are the eigenvectors of the covariance matrix and the diagonal elements of $\sum$ are the square roots of the eigenvalues of the covariance matrix. Calculating SVD is beyond the scope of this chapter; however, eigenvectors found using SVD should be similar to those derived from a covariance matrix.

# Using PCA to visualize high-dimensional data

It is easy to discover patterns by visualizing data with two or three dimensions. A high-dimensional dataset cannot be represented graphically, but we can still gain some insights into its structure by reducing it to two or three principal components.

Collected in 1936, Fisher's Iris data set is a collection of fifty samples from each of the three species of Iris: Iris setosa, Iris virginica, and Iris versicolor. The explanatory variables are measurements of the length and width of the petals and sepals of the flowers. The Iris dataset is commonly used to test classification models, and is included with scikit-learn. Let's reduce the `iris` dataset's four dimensions so that we can visualize it in two dimensions:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.decomposition import PCA
>>> from sklearn.datasets import load_iris
```

First, we load the built-in iris data set and instantiate a `PCA` estimator. The `PCA` class takes a number of principal components to retain as a hyperparameter. Like the other estimators, `PCA` exposes a `fit_transform()` method that returns the reduced data matrix:

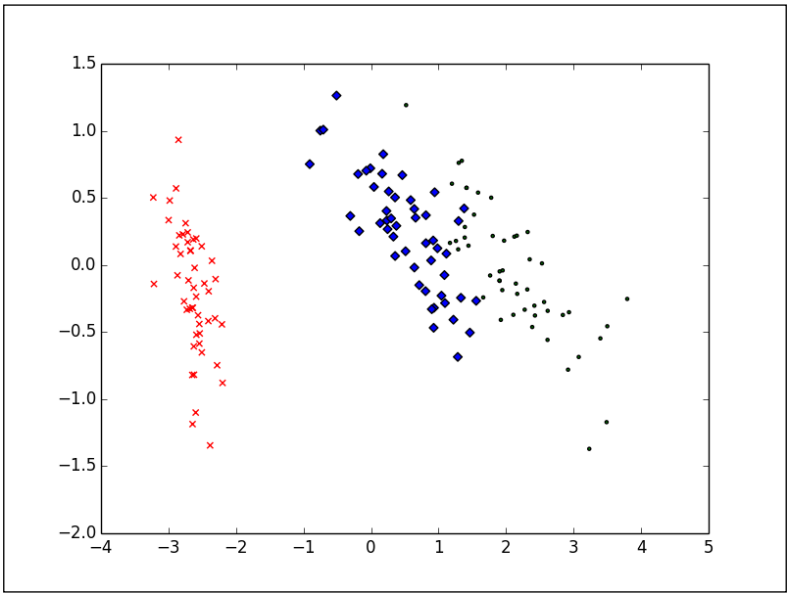
```
>>> data = load_iris()
>>> y = data.target
>>> X = data.data
>>> pca = PCA(n_components=2)
>>> reduced_X = pca.fit_transform(X)
```

Finally, we assemble and plot the reduced data:

```
>>> red_x, red_y = [], []
>>> blue_x, blue_y = [], []
>>> green_x, green_y = [], []
>>> for i in range(len(reduced_X)):
>>>     if y[i] == 0:
>>>         red_x.append(reduced_X[i][0])
>>>         red_y.append(reduced_X[i][1])
>>>     elif y[i] == 1:
>>>         blue_x.append(reduced_X[i][0])
>>>         blue_y.append(reduced_X[i][1])
>>>     else:
>>>         green_x.append(reduced_X[i][0])
```

```
>>>         green_y.append(reduced_X[i][1])
>>> plt.scatter(red_x, red_y, c='r', marker='x')
>>> plt.scatter(blue_x, blue_y, c='b', marker='D')
>>> plt.scatter(green_x, green_y, c='g', marker='.')
>>> plt.show()
```

The reduced instances are plotted in the following figure. Each of the dataset's three classes is indicated by its own marker style. From this two-dimensional view of the data, it is clear that one of the classes can be easily separated from the other two overlapping classes. It would be difficult to notice this structure without a graphical representation. This insight can inform our choice of classification model.




# Face recognition with PCA

Now let's apply PCA to a face-recognition problem. Face recognition is the supervised classification task of identifying a person from an image of his or her face. In this example, we will use a data set called *Our Database of Faces* from AT&T Laboratories, Cambridge. The data set contains ten images each of forty people. The images were created under different lighting conditions, and the subjects varied their facial expressions. The images are gray scale and 92 x 112 pixels in dimension. The following is an example image:

While these images are small, a feature vector that encodes the intensity of every pixel will have 10,304 dimensions. Training from such high-dimensional data could require many samples to avoid over-fitting. Instead, we will use PCA to compactly represent the images in terms of a small number of principal components.

We can reshape the matrix of pixel intensities for an image into a vector, and create a matrix of these vectors for all of the training images. Each image is a linear combination of this data set's principal components. In the context of face recognition, these principal components are called **eigenfaces**. The eigenfaces can be thought of as standardized components of faces. Each face in the data set can be expressed as some combination of the eigenfaces, and can be approximated as a combination of the most important eigenfaces:

```
>>> from os import walk, path
>>> import numpy as np
>>> import mahotas as mh
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.preprocessing import scale
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.metrics import classification_report
>>> X = []
>>> y = []
```

We begin by loading the images into `NumPy` arrays, and reshaping their matrices into vectors:

```
>>> for dir_path, dir_names, file_names in walk('data/att-faces/orl_
faces'):
>>>     for fn in file_names:
>>>         if fn[-3:] == 'pgm':
>>>             image_filename = path.join(dir_path, fn)
>>>             X.append(scale(mh.imread(image_filename, as_
grey=True).reshape(10304).astype('float32')))
>>>             y.append(dir_path)
>>> X = np.array(X)
```

We then randomly split the images into training and test sets, and fit the `PCA` object on the training set:

```
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)
>>> pca = PCA(n_components=150)
```

We reduce all of the instances to 150 dimensions and train a logistic regression classifier. The data set contains forty classes; scikit-learn automatically creates binary classifiers using the one versus all strategy behind the scenes:

```
>>> X_train_reduced = pca.fit_transform(X_train)
>>> X_test_reduced = pca.transform(X_test)
>>> print 'The original dimensions of the training data were', X_
train.shape
>>> print 'The reduced dimensions of the training data are', X_train_
reduced.shape
>>> classifier = LogisticRegression()
>>> accuracies = cross_val_score(classifier, X_train_reduced, y_train)
```

Finally, we evaluate the performance of the classifier using cross-validation and a test set. The average per-class F1 score of the classifier trained on the full data was 0.94, but required significantly more time to train and could be prohibitively slow in an application with more training instances:

```
>>> print 'Cross validation accuracy:', np.mean(accuracies),
accuracies
>>> classifier.fit(X_train_reduced, y_train)
>>> predictions = classifier.predict(X_test_reduced)
>>> print classification_report(y_test, predictions)
```

The following is the output of the script:

```
The original dimensions of the training data were (300, 10304)
The reduced dimensions of the training data are (300, 150)
Cross validation accuracy: 0.833841819347 [ 0.82882883  0.83
0.84269663]
             precision    recall  f1-score   support

data/att-faces/orl_faces/s1       1.00      1.00      1.00         2
data/att-faces/orl_faces/s10       1.00      1.00      1.00         2
data/att-faces/orl_faces/s11       1.00      0.60      0.75         5
...
data/att-faces/orl_faces/s9       1.00      1.00      1.00         2

avg / total       0.92      0.89      0.89       100
```

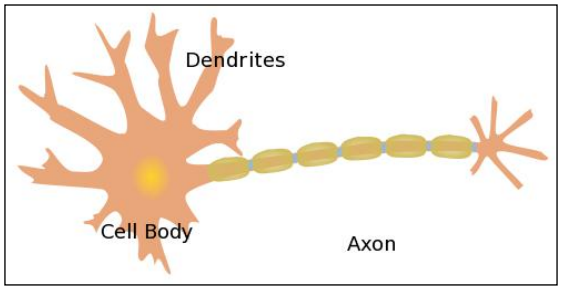
# Summary

In this chapter, we examined the problem of dimensionality reduction. High-dimensional data cannot be visualized easily. High-dimensional data sets may also suffer from the curse of dimensionality; estimators require many samples to learn to generalize from high-dimensional data. We mitigated these problems using a technique called principal component analysis, which reduces a high-dimensional, possibly-correlated data set to a lower-dimensional set of uncorrelated principal components by projecting the data onto a lower-dimensional subspace. We used principal component analysis to visualize the four-dimensional Iris data set in two dimensions, and build a face-recognition system. In the next chapter, we will return to supervised learning. We will discuss an early classification algorithm called the perceptron, which will prepare us to discuss more advanced models in the last few chapters.

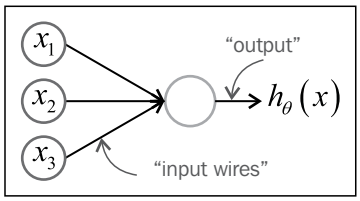
# 8
# The Perceptron

In previous chapters we discussed generalized linear models that relate a linear combination of explanatory variables and model parameters to a response variable using a link function. In this chapter, we will discuss another linear model called the perceptron. The perceptron is a binary classifier that can learn from individual training instances, which can be useful for training from large datasets. More importantly, the perceptron and its limitations inspire the models that we will discuss in the final chapters.

Invented by Frank Rosenblatt at the Cornell Aeronautical Laboratory in the late 1950's, the development of the perceptron was originally motivated by efforts to simulate the human brain. A brain is composed of cells called **neurons** that process information and connections between neurons called **synapses** through which information is transmitted. It is estimated that human brain is composed of as many as 100 billion neurons and 100 trillion synapses. As shown in the following image, the main components of a neuron are dendrites, a body, and an axon. The dendrites receive electrical signals from other neurons. The signals are processed in the neuron's body, which then sends a signal through the axon to another neuron.



An individual neuron can be thought of as a computational unit that processes one or more inputs to produce an output. A perceptron functions analogously to a neuron; it accepts one or more inputs, processes them, and returns an output. It may seem that a model of just one of the hundreds of billions of neurons in the human brain will be of limited use. To an extent that is true; the perceptron cannot approximate some basic functions. However, we will still discuss perceptrons for two reasons. First, perceptrons are capable of online, error-driven learning; the learning algorithm can update the model's parameters using a single training instance rather than the entire batch of training instances. Online learning is useful for learning from training sets that are too large to be represented in memory. Second, understanding how the perceptron works is necessary to understand some of the more powerful models that we will discuss in subsequent chapters, including support vector machines and artificial neural networks. Perceptrons are commonly visualized using a diagram like the following one:




The circles labeled $x_1$, $x_2$, and $x_3$ are inputs units. Each input unit represents one feature. Perceptrons frequently use an additional input unit that represents a constant bias term, but this input unit is usually omitted from diagrams. The circle in the center is a computational unit or the neuron's body. The edges connecting the input units to the computational unit are analogous to dendrites. Each edge is **weighted**, or associated with a parameter. The parameters can be interpreted easily; an explanatory variable that is correlated with the positive class will have a positive weight, and an explanatory variable that is correlated with the negative class will have a negative weight. The edge directed away from the computational unit returns the output and can be thought of as the axon.

# Activation functions

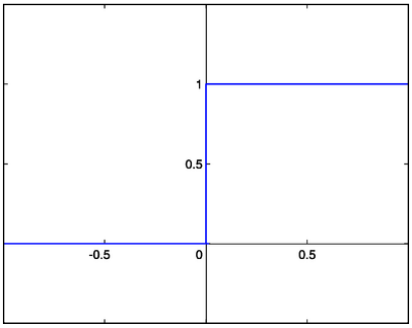
The perceptron classifies instances by processing a linear combination of the explanatory variables and the model parameters using an **activation function** as shown in the following equation. The linear combination of the parameters and inputs is sometimes called the perceptron's **preactivation**.

$$y = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Here, $w_i$ are the model's parameters, $b$ is a constant bias term, and $\phi$ is the activation function. Several different activation functions are commonly used. Rosenblatt's original perceptron used the **Heaviside step** function. Also called the unit step function, the Heaviside step function is shown in the following equation, where $x$ is the weighted combination of the features:

$$g(x) = \begin{cases} 1, & \text{if } x>0 \\ 0, & \text{elsewhere} \end{cases}$$

If the weighted sum of the explanatory variables and the bias term is greater than zero, the activation function returns one and the perceptron predicts that the instance is the positive class. Otherwise, the function returns zero and the perceptron predicts that the instance is the negative class. The Heaviside step activation function is plotted in the following figure:



Another common activation function is the **logistic sigmoid** activation function. The gradients for this activation function can be calculated efficiently, which will be important in later chapters when we construct artificial neural networks. The logistic sigmoid activation function is given by the following equation, where $x$ is the sum of the weighted inputs:

$$g(x) = \frac{1}{1 + e^{-x}}$$

This model should seem familiar; it is a linear combination of the values of the explanatory variables and the model parameters processed through the logistic function. That is, this is identical to the model for logistic regression. While a perceptron with a logistic sigmoid activation function has the same model as logistic regression, it learns its parameters differently.

# The perceptron learning algorithm

The perceptron learning algorithm begins by setting the weights to zero or to small random values. It then predicts the class for a training instance. The perceptron is an **error-driven** learning algorithm; if the prediction is correct, the algorithm continues to the next instance. If the prediction is incorrect, the algorithm updates the weights. More formally, the update rule is given by the following:

$$w_i(t+1) = w_i(t) + \alpha\left(d_j - y_j(t)\right)x_{j,i}, \text{for all feature } 0 \le i \le n$$

For each training instance, the value of the parameter for each explanatory variable is incremented by $\alpha\left(d_j - y_j(t)\right)x_{j,i}$, where $d_j$ is the true class for instance $j$, $y_j(t)$ is the predicted class for instance $j$, $x_{j,i}$ is the value of the $ith$ explanatory variable for instance $j$, and $\alpha$ is a hyperparameter that controls the learning rate. If the prediction is correct, $d_j - y_j(t)$ equals zero, and the $\alpha\left(d_j - y_j(t)\right)x_{j,i}$ term equals zero. So, if the prediction is correct, the weight is not updated. If the prediction is incorrect, the weight is incremented by the product of the learning rate, $d_j - y_j(t)$, and the value of the feature.

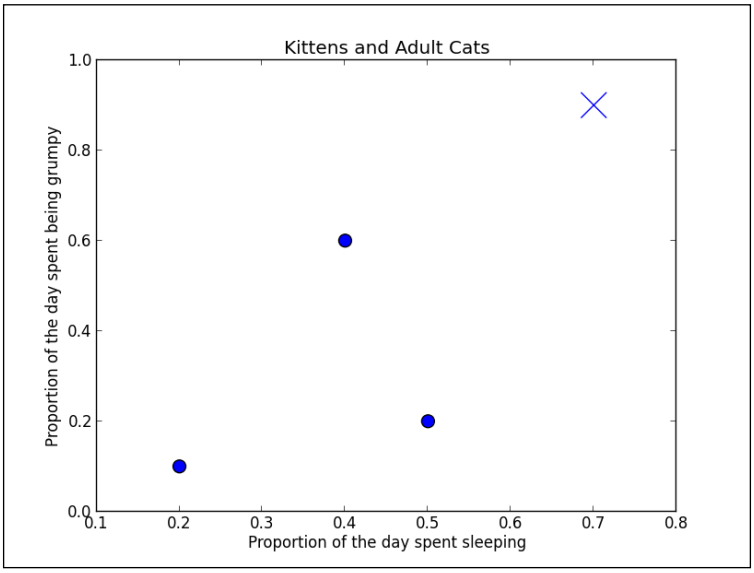
This update rule is similar to the update rule for gradient descent in that the weights are adjusted towards classifying the instance correctly and the size of the update is controlled by a learning rate. Each pass through the training instances is called an **epoch.** The learning algorithm has converged when it completes an epoch without misclassifying any of the instances. The learning algorithm is not guaranteed to converge; later in this chapter, we will discuss linearly inseparable datasets for which convergence is impossible. For this reason, the learning algorithm also requires a hyperparameter that specifies the maximum number of epochs that can be completed before the algorithm terminates.

# Binary classification with the perceptron

Let's work through a toy classification problem. Suppose that you wish to separate adult cats from kittens. Only two explanatory variables are available in your dataset: the proportion of the day that the animal was asleep and the proportion of the day that the animal was grumpy. Our training data consists of the following four instances:

| Instance | Proportion of the day spent sleeping | Proportion of the day spent being grumpy | Kitten or Adult? |
|---|---|---|---|
| 1 | 0.2 | 0.1 | Kitten |
| 2 | 0.4 | 0.6 | Kitten |
| 3 | 0.5 | 0.2 | Kitten |
| 4 | 0.7 | 0.9 | Adult |

The following scatter plot of the instances confirms that they are linearly separable:



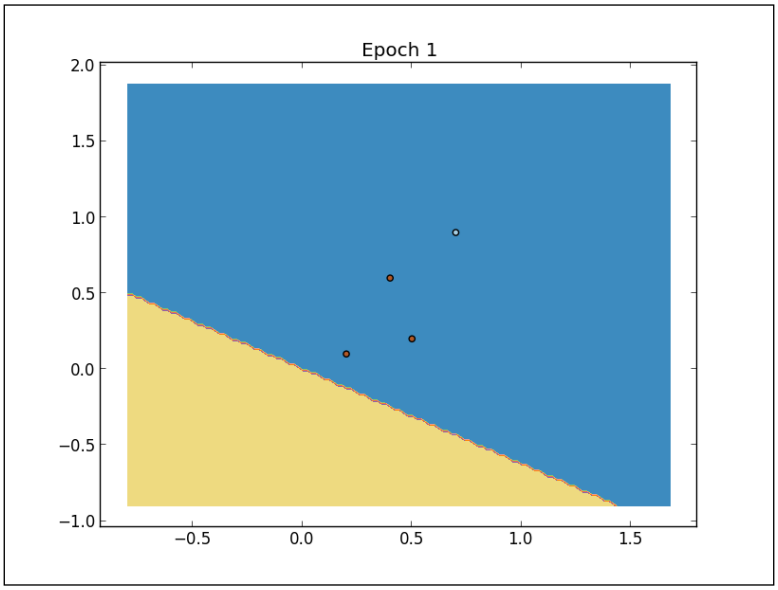
Our goal is to train a perceptron that can classify animals using the two real-valued explanatory variables. We will represent kittens with the positive class and adult cats with the negative class. The preceding network diagram describes the perceptron that we will train.

Our perceptron has three input units. $x_1$ is the input unit for the bias term. $x_2$ and $x_3$ are input units for the two features. Our perceptron's computational unit uses a Heaviside activation function. In this example, we will set the maximum number of training epochs to ten; if the algorithm does not converge within 10 epochs, it will stop and return the current values of the weights. For simplicity, we will set the learning rate to one. Initially, we will set all of the weights to zero. Let's examine the first training epoch, which is shown in the following table:

| **Epoch 1** | | | | |
|---|---|---|---|---|
| **Instance** | **Initial Weights**<br>**x**<br>**Activation** | **Prediction, Target** | **Correct** | **Updated weights** |
| 0 | 0, 0, 0;<br>1.0, 0.2, 0.1;<br>1.0*0 + 0.2*0 + 0.1*0 = 0.0; | 0, 1 | False | 1.0, 0.2, 0.1 |
| 1 | 1.0, 0.2, 0.1;<br>1.0, 0.4, 0.6;<br>1.0*1.0 + 0.4*0.2 + 0.6*0.1 = 1.14; | 1, 1 | True | 1.0, 0.2, 0.1 |
| 2 | 1.0, 0.2, 0.1;<br>1.0, 0.5, 0.2;<br>1.0*1.0 + 0.5*0.2 + 0.2*0.1 = 1.12; | 1, 1 | True | 1.0, 0.2, 0.1 |
| 3 | 1.0, 0.2, 0.1;<br>1.0, 0.7, 0.9;<br>1.0*1.0 + 0.7*0.2 + 0.9*0.1 = 1.23; | 1, 0 | False | 0, -0.5, -0.8 |

Initially, all of the weights are equal to zero. The weighted sum of the explanatory variables for the first instance is zero, the activation function outputs zero, and the perceptron incorrectly predicts that the kitten is an adult cat. As the prediction was incorrect, we update the weights according to the update rule. We increment each of the weights by the product of the learning rate, the difference between the true and predicted labels and the value of the corresponding feature.

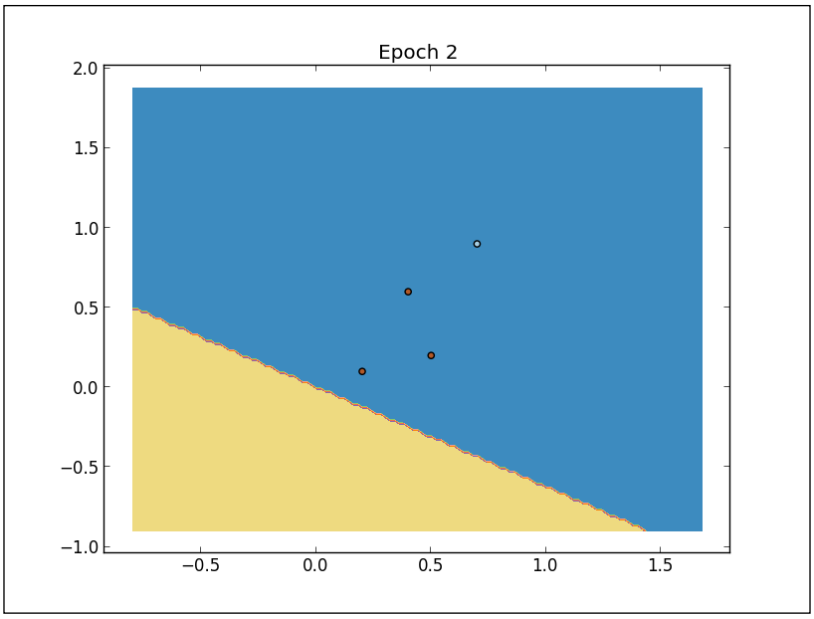
We then continue to the second training instance and calculate the weighted sum of its features using the updated weights. This sum equals 1.14, so the activation function outputs one. This prediction is correct, so we continue to the third training instance without updating the weights. The prediction for the third instance is also correct, so we continue to the fourth training instance. The weighted sum of the features for the fourth instance is 1.23. The activation function outputs one, incorrectly predicting that this adult cat is a kitten. Since this prediction is incorrect, we increment each weight by the product of the learning rate, the difference between the true and predicted labels, and its corresponding feature. We completed the first epoch by classifying all of the instances in the training set. The perceptron did not converge; it classified half of the training instances incorrectly. The following figure depicts the decision boundary after the first epoch:



Note that the decision boundary moved throughout the epoch; the decision boundary formed by the weights at the end of the epoch would not necessarily have produced the same predictions seen earlier in the epoch. Since we have not exceeded the maximum number of training epochs, we will iterate through the instances again. The second training epoch is shown in the following table:

| **Epoch 2** | | | | |
|---|---|---|---|---|
| **Instance** | **Initial Weights**<br>**x**<br>**Activation** | **Prediction, Target** | **Correct** | **Updated weights** |
| 0 | 0, -0.5, -0.8<br>1.0, 0.2, 0.1<br>1.0*0 + 0.2*-0.5 + 0.1*-0.8 = -0.18 | 0, 1 | False | 1, -0.3, -0.7 |
| 1 | 1, -0.3, -0.7<br>1.0, 0.4, 0.6<br>1.0*1.0 + 0.4*-0.3 + 0.6*-0.7 = 0.46 | 1, 1 | True | 1, -0.3, -0.7 |
| 2 | 1, -0.3, -0.7<br>1.0, 0.5, 0.2<br>1.0*1.0 + 0.5*-0.3 + 0.2*-0.7 = 0.71 | 1, 1 | True | 1, -0.3, -0.7 |
| 3 | 1, -0.3, -0.7<br>1.0, 0.7, 0.9<br>1.0*1.0 + 0.7*-0.3 + 0.9*-0.7 = 0.16 | 1, 0 | False | 0, -1, -1.6 |

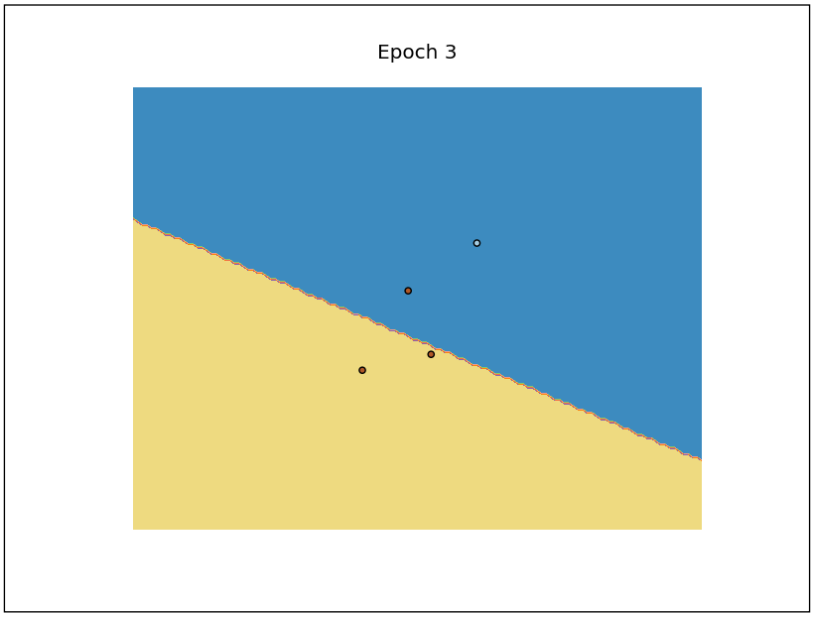
The second epoch begins using the values of the weights from the first epoch. Two training instances are classified incorrectly during this epoch. The weights are updated twice, but the decision boundary at the end of the second epoch is similar the decision boundary at the end of the first epoch.




The algorithm failed to converge during this epoch, so we will continue training. The following table describes the third training epoch:

| **Epoch 3** | | | | |
|---|---|---|---|---|
| **Instance** | **Initial Weights**<br>**x**<br>**Activation** | **Prediction, Target** | **Correct** | **Updated Weights** |
| 0 | 0, -1, -1.6<br>1.0, 0.2, 0.1<br>1.0*0 + 0.2*-1.0 + 0.1*-1.6 = -0.36 | 0, 1 | `False` | 1,-0.8, -1.5 |
| 1 | 1,-0.8, -1.5<br>1.0, 0.4, 0.6<br>1.0*1.0 + 0.4*-0.8 + 0.6*-1.5 = -0.22 | 0, 1 | False | 2, -0.4, -0.9 |
| 2 | 2, -0.4, -0.9<br>1.0, 0.5, 0.2<br>1.0*2.0 + 0.5*-0.4 + 0.2*-0.9 = 1.62 | 1, 1 | True | 2, -0.4, -0.9 |
| 3 | 2, -0.4, -0.9<br>1.0, 0.7, 0.9<br>1.0*2.0 + 0.7*-0.4 + 0.9*-0.9 = 0.91 | 1, 0 | False | 1, -1.1, -1.8 |

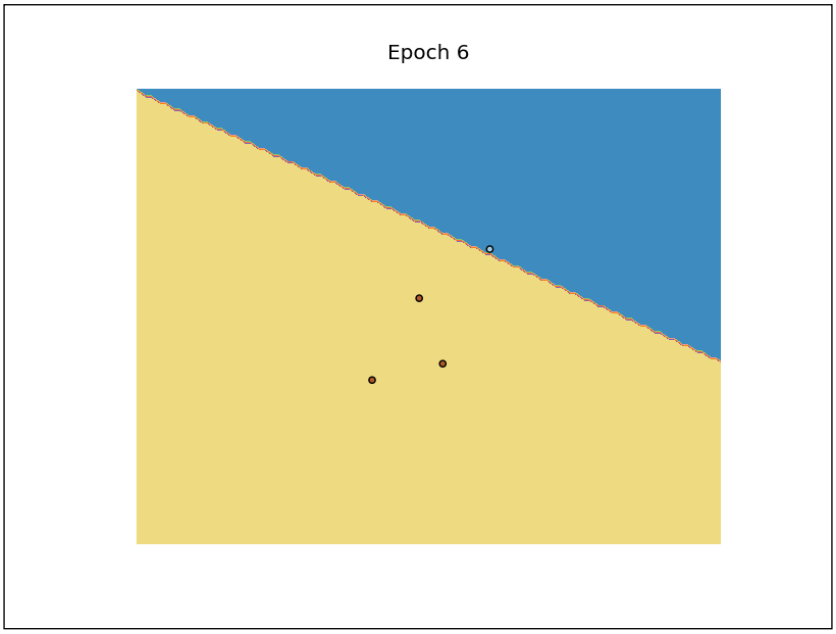
The perceptron classified more instances incorrectly during this epoch than during previous epochs. The following figure depicts the decision boundary at the end of the third epoch:



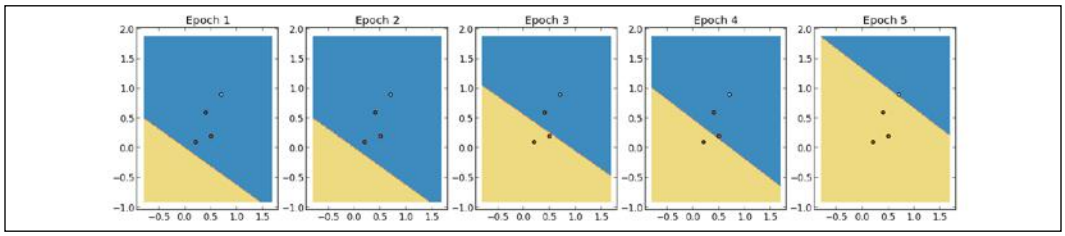

The perceptron continues to update its weights throughout the fourth and fifth training epochs, and it continues to classify training instances incorrectly. During the sixth epoch the perceptron classified all of the instances correctly; it converged on a set of weights that separates the two classes. The following table describes the sixth training epoch:

| **Epoch 6** | | | | |
|---|---|---|---|---|
| **Instance** | **Initial Weights**<br>**x**<br>**Activation** | **Prediction, Target** | **Correct** | **Updated weights** |
| 0 | 2, -1, -1.5<br>1.0, 0.2, 0.1<br>1.0*2 + 0.2*-1 + 0.1*-1.5 = 1.65 | 1, 1 | True | 2, -1, -1.5 |
| 1 | 2, -1, -1.5<br>1.0, 0.4, 0.6<br>1.0*2 + 0.4*-1 + 0.6*-1.5 = 0.70 | 1, 1 | True | 2, -1, -1.5 |
| 2 | 2, -1, -1.5<br>1.0, 0.5, 0.2<br>1.0*2 + 0.5*-1 + 0.2*-1.5 = 1.2 | 1, 1 | True | 2, -1, -1.5 |
| 3 | 2, -1, -1.5<br>1.0, 0.7, 0.9<br>1.0*2 + 0.7*-1 + 0.9*-1.5 = -0.05 | 0, 0 | True | 2, -1, -1.5 |

The decision boundary at the end of the sixth training epoch is shown in the following figure:



The following figure shows the decision boundary throughout all the training epochs.




# Document classification with the perceptron

scikit-learn provides an implementation of the perceptron. As with the other implementations that we used, the constructor for the `Perceptron` class accepts keyword arguments that set the algorithm's hyperparameters. `Perceptron` similarly exposes the `fit_transform()` and `predict()` methods. `Perceptron` also provides a `partial_fit()` method, which allows the classifier to train and make predictions for streaming data.

In this example, we train a perceptron to classify documents from the 20 newsgroups dataset. The dataset consists of approximately 20,000 documents sampled from 20 Usenet newsgroups. The dataset is commonly used in document classification and clustering experiments; scikit-learn provides a convenience function to download and read the dataset. We will train a perceptron to classify documents from three newsgroups: `rec.sports.hockey`, `rec.sports.baseball`, and `rec.auto`. scikit-learn's `Perceptron` natively supports multiclass classification; it will use the one versus all strategy to train a classifier for each of the classes in the training data. We will represent the documents as TF-IDF-weighted bags of words. The `partial_fit()` method could be used in conjunction with `HashingVectorizer` to train from large or streaming data in a memory-constrained setting:

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> from sklearn.metrics.metrics import f1_score, classification_
report
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from sklearn.linear_model import Perceptron

>>> categories = ['rec.sport.hockey', 'rec.sport.baseball', 'rec.
autos']
>>> newsgroups_train = fetch_20newsgroups(subset='train',
categories=categories, remove=('headers', 'footers', 'quotes'))
```

```
>>> newsgroups_test = fetch_20newsgroups(subset='test',
categories=categories, remove=('headers', 'footers', 'quotes'))

>>> vectorizer = TfidfVectorizer()
>>> X_train = vectorizer.fit_transform(newsgroups_train.data)
>>> X_test = vectorizer.transform(newsgroups_test.data)

>>> classifier = Perceptron(n_iter=100, eta0=0.1)
>>> classifier.fit_transform(X_train, newsgroups_train.target )
>>> predictions = classifier.predict(X_test)
>>> print classification_report(newsgroups_test.target, predictions)
```

The following is the output of the script:

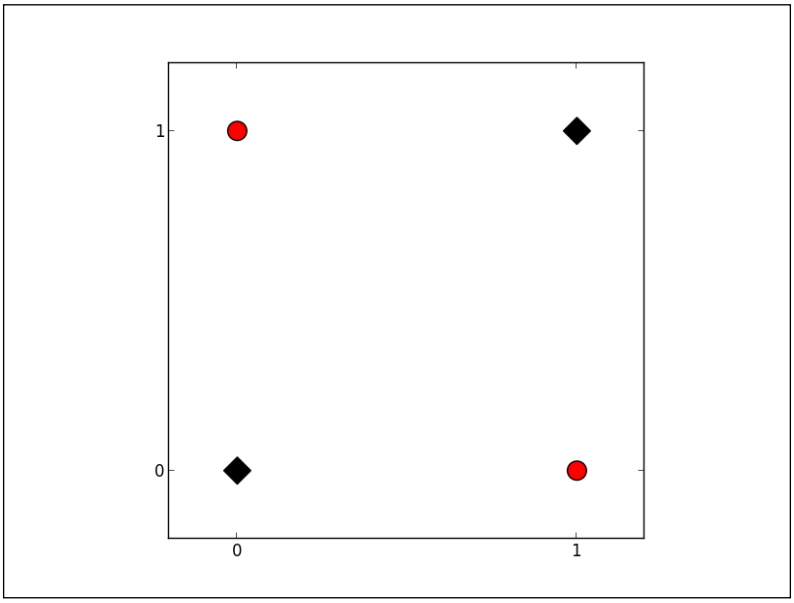
```
                precision    recall  f1-score   support

             0       0.89      0.87      0.88       396
             1       0.87      0.78      0.82       397
             2       0.79      0.88      0.83       399

avg / total       0.85      0.85      0.85      1192
```

First, we download and read the dataset using the `fetch_20newsgroups()` function. Consistent with other built-in datasets, the function returns an object with `data`, `target`, and `target_names` fields. We also specify that the documents' headers, footers, and quotes should be removed. Each of the newsgroups used different conventions in the headers and footers; retaining these explanatory variables makes classifying the documents artificially easy. We produce TF-IDF vectors using `TfifdVectorizer`, train the perceptron, and evaluate it on the test set. Without hyperparameter optimization, the perceptron's average precision, recall, and F1 score are 0.85.

# Limitations of the perceptron

While the perceptron classified the instances in our example well, the model has limitations. Linear models like the perceptron with a Heaviside activation function are not **universal function approximators**; they cannot represent some functions. Specifically, linear models can only learn to approximate the functions for **linearly separable** datasets. The linear classifiers that we have examined find a hyperplane that separates the positive classes from the negative classes; if no hyperplane exists that can separate the classes, the problem is not linearly separable.

A simple example of a function that is linearly inseparable is the logical operation **XOR**, or exclusive disjunction. The output of XOR is one when one of its inputs is equal to one and the other is equal to zero. The inputs and outputs of XOR are plotted in two dimensions in the following graph. When XOR outputs **1**, the instance is marked with a circle; when XOR outputs **0**, the instance is marked with a diamond, as shown in the following figure:




It is impossible to separate the circles from the diamonds using a single straight line. Suppose that the instances are pegs on a board. If you were to stretch a rubber band around both of the positive instances, and stretch a second rubber band around both of the negative instances, the bands would intersect in the middle of the board. The rubber bands represent **convex hulls**, or the envelope that contains all of the points within the set and all of the points along any line connecting a pair points within the set. Feature representations are more likely to be linearly separable in higher dimensional spaces than lower dimensional spaces. For instance, text classification problems tend to be linearly separable when high-dimensional representations like the bag-of-words are used.

In the next two chapters, we will discuss techniques that can be used to model linearly inseparable data. The first technique, called **kernelization**, projects linearly inseparable data to a higher dimensional space in which it is linearly separable. Kernelization can be used in many models, including perceptrons, but it is particularly associated with support vector machines, which we will discuss in the next chapter. Support vector machines also support techniques that can find the hyperplane that separates linearly inseparable classes with the fewest errors. The second technique creates a directed graph of perceptrons. The resulting model, called an **artificial neural network**, is a universal function approximator; we will discuss artificial neural networks in *Chapter 10*, *From the Perceptron to Artificial Neural Networks*.

# Summary

In this chapter, we discussed the perceptron. Inspired by neurons, the perceptron is a linear model for binary classification. The perceptron classifies instances by processing a linear combination of the explanatory variables and weights with an activation function. While a perceptron with a logistic sigmoid activation function is the same model as logistic regression, the perceptron learns its weights using an online, error-driven algorithm. The perceptron can be used effectively in some problems. Like the other linear classifiers that we have discussed, the perceptron is not a universal function approximator; it can only separate the instances of one class from the instances of the other using a hyperplane. Some datasets are not linearly separable; that is, no possible hyperplane can classify all of the instances correctly. In the following chapters, we will discuss two models that can be used with linearly inseparable data: the artificial neural network, which creates a universal function approximator from a graph of perceptrons and the support vector machine, which projects the data onto a higher dimensional space in which it is linearly separable.

# 9
# From the Perceptron to Support Vector Machines

In the previous chapter we discussed the perceptron. As a binary classifier, the perceptron cannot be used to effectively classify linearly inseparable feature representations. We encountered a similar problem to this in our discussion of multiple linear regression in *Chapter 2*, *Linear Regression*; we examined a dataset in which the response variable was not linearly related to the explanatory variables. To improve the accuracy of the model, we introduced a special case of multiple linear regression called polynomial regression. We created synthetic combinations of features, and were able to model a linear relationship between the response variable and the features in the higher-dimensional feature space.

While this method of increasing the dimensions of the feature space may seem like a promising technique to use when approximating nonlinear functions with linear models, it suffers from two related problems. The first is a computational problem; computing the mapped features and working with larger vectors requires more computing power. The second problem pertains to generalization; increasing the dimensions of the feature representation introduces the curse of dimensionality. Learning from high-dimensional feature representations requires exponentially more training data to avoid overfitting.

In this chapter, we will discuss a powerful model for classification and regression called the **support vector machine** (**SVM**). First, we will revisit mapping features to higher-dimensional spaces. Then, we will discuss how support vector machines mitigate the computation and generalization problems encountered when learning from the data mapped to higher-dimensional spaces. Entire books are devoted to describing support vector machines, and describing the optimization algorithms used to train SVMs requires more advanced math than we have used in previous chapters. Instead of working through toy examples in detail as we have done in previous chapters, we will try to develop an intuition for how support vector machines work in order to apply them effectively with scikit-learn.

# Kernels and the kernel trick

Recall that the perceptron separates the instances of the positive class from the instances of the negative class using a hyperplane as a decision boundary. The decision boundary is given by the following equation:

$$f(x) = \langle w, x \rangle + b$$

Predictions are made using the following function:

$$h(x) = \text{sign}(f(x))$$

Note that previously we expressed the inner product $\langle w, x \rangle$ as $w^T x$. To be consistent with the notational conventions used for support vector machines, we will adopt the former notation in this chapter.

While the proof is beyond the scope of this chapter, we can write the model differently. The following expression of the model is called the **dual** form. The expression we used previously is the **primal** form:

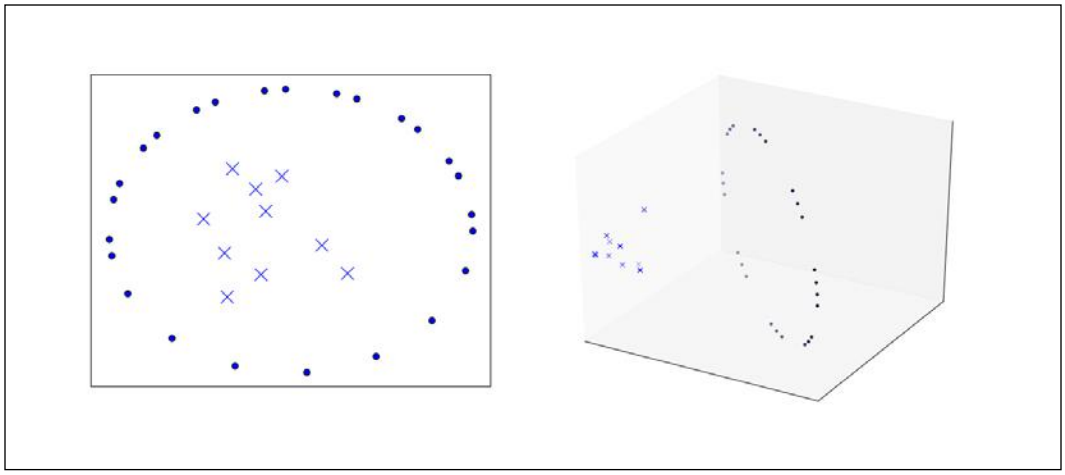
$$f(x) = \langle w, x \rangle + b = \sum \alpha_i y_i \langle x_i, x \rangle + b$$

The most important difference between the primal and dual forms is that the primal form computes the inner product of the *model parameters* and the test instance's feature vector, while the dual form computes the inner product of the *training instances* and the test instance's feature vector. Shortly, we will exploit this property of the dual form to work with linearly inseparable classes. First, we must formalize our definition of mapping features to higher-dimensional spaces.

In the section on polynomial regression in *Chapter 2*, *Linear Regression*, we mapped features to a higher-dimensional space in which they were linearly related to the response variable. The mapping increased the number of features by creating quadratic terms from combinations of the original features. These synthetic features allowed us to express a nonlinear function with a linear model. In general, a mapping is given by the following expression:

$$x \rightarrow \phi(x)$$
$$\phi : R^d \rightarrow R^D$$

The plot on the left in the following figure shows the original feature space of a linearly inseparable data set. The plot on the right shows that the data is linearly separable after mapping to a higher-dimensional space:



Let's return to the dual form of our decision boundary, and the observation that the feature vectors appear only inside of a dot product. We could map the data to a higher-dimensional space by applying the mapping to the feature vectors as follows:

$$f(x) = \sum \alpha_i y_i \langle x_i, x \rangle + b$$

$$f(x) = \sum \alpha_i y_i \langle \phi(x_i) \phi(x) \rangle + b$$

As noted, this mapping allows us to express more complex models, but it introduces computation and generalization problems. Mapping the feature vectors and computing their dot products can require a prohibitively large amount of processing power.

Observe in the second equation that while we have mapped the feature vectors to a higher-dimensional space, the feature vectors still only appear as a dot product. The dot product is scalar; we do not require the mapped feature vectors once this scalar has been computed. If we can use a different method to produce the same scalar as the dot product of the mapped vectors, we can avoid the costly work of explicitly computing the dot product and mapping the feature vectors.

Fortunately, there is such a method called the **kernel trick**. A **kernel** is a function that, given the original feature vectors, returns the same value as the dot product of its corresponding mapped feature vectors. Kernels do not explicitly map the feature vectors to a higher-dimensional space, or calculate the dot product of the mapped vectors. Kernels produce the same value through a different series of operations that can often be computed more efficiently. Kernels are defined more formally in the following equation:

$$K(x,z) = \langle \phi(x), \phi(z) \rangle$$

Let's demonstrate how kernels work. Suppose that we have two feature vectors, $x$ and $z$:

$$x = (x_1, x_2)$$

$$z = (z_1, z_2)$$

In our model, we wish to map the feature vectors to a higher-dimensional space using the following transformation:

$$\phi(x) = x^2$$

The dot product of the mapped, normalized feature vectors is equivalent to:

$$\langle \phi(x), \phi(z) \rangle = \left\langle \left(x_1^2, x_2^2, \sqrt{2}x_1x_2\right), \left(z_1^2, z_2^2, \sqrt{2}z_1z_2\right) \right\rangle$$

The kernel given by the following equation produces the same value as the dot product of the mapped feature vectors:

$$K(x,z) = \langle x, z \rangle^2 = (x_1z_1 + x_2z_2)^2 = x_1^2z_1^2 + 2x_1z_1x_2z_2 + x_2^2z_2^2$$

$$K(x,z) = \langle \phi(x), \phi(z) \rangle$$

Let's plug in values for the feature vectors to make this example more concrete:

$$x = (4,9)$$

$$z = (3,3)$$

$$K(x,z) = 4^2 * 3^2 + 2*4*3*9*3 = 1521$$

$$\langle \phi(x), \phi(z) \rangle = \langle (4^2, 9^2, \sqrt{2}*4*9), (3^2, 3^2, \sqrt{2}*3*3) \rangle = 1521$$

The kernel $K(x,z)$ produced the same value as the dot product $\langle \phi(x), \phi(z) \rangle$ of the mapped feature vectors, but never explicitly mapped the feature vectors to the higher-dimensional space and required fewer arithmetic operations. This example used only two dimensional feature vectors. Data sets with even a modest number of features can result in mapped feature spaces with massive dimensions. scikit-learn provides several commonly used kernels, including the polynomial, sigmoid, Gaussian, and linear kernels. Polynomial kernels are given by the following equation:

$$K(x,x') = (1 + x \times x')^k$$

Quadratic kernels, or polynomial kernels where *k* is equal to 2, are commonly used in natural language processing.

The sigmoid kernel is given by the following equation. $\gamma$ and $r$ are hyperparameters that can be tuned through cross-validation:

$$K(x,x') = \tanh \langle \gamma(x,x') + r \rangle$$

The Gaussian kernel is a good first choice for problems requiring nonlinear models. The Gaussian kernel is a **radial basis function**. A decision boundary that is a hyperplane in the mapped feature space is similar to a decision boundary that is a hypersphere in the original space. The feature space produced by the Gaussian kernel can have an infinite number of dimensions, a feat that would be impossible otherwise. The Gaussian kernel is given by the following equation:

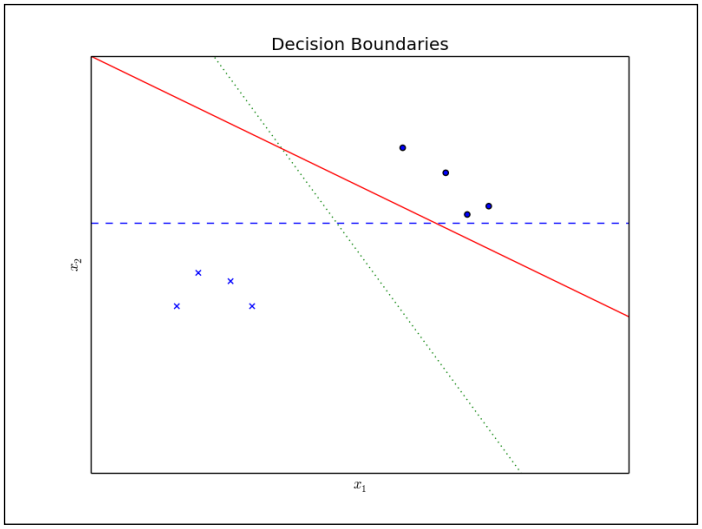
$$K(x,x') = e\left(-\|x - x'\|^2 / \sigma^2\right)$$

$\gamma$ is a hyperparameter. It is always important to scale the features when using support vector machines, but feature scaling is especially important when using the Gaussian kernel.

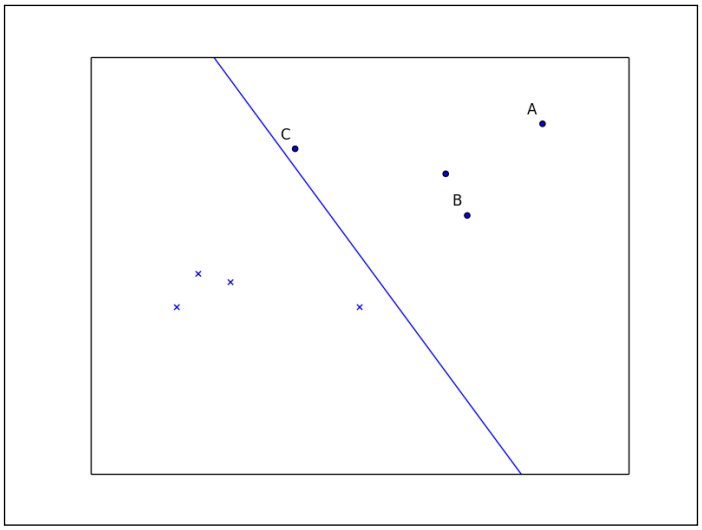
Choosing a kernel can be challenging. Ideally, a kernel will measure the similarity between instances in a way that is useful to the task. While kernels are commonly used with support vector machines, they can also be used with any model that can be expressed in terms of the dot product of two feature vectors, including logistic regression, perceptrons, and principal component analysis. In the next section, we will address the second problem caused by mapping to high-dimensional feature spaces: generalization.

# Maximum margin classification and support vectors

The following figure depicts instances from two linearly separable classes and three possible decision boundaries. All of the decision boundaries separate the training instances of the positive class from the training instances of the negative class, and a perceptron could learn any of them. Which of these decision boundaries is most likely to perform best on test data?



From this visualization, it is intuitive that the dotted decision boundary is the best. The solid decision boundary is near many of the positive instances. The test set could contain a positive instance that has a slightly smaller value for the first explanatory variable, $x_1$; this instance would be classified incorrectly. The dashed decision boundary is farther away from most of the training instances; however, it is near one of the positive instances and one of the negative instances. The following figure provides a different perspective on evaluating decision boundaries:




Assume that the line plotted is the decision boundary for a logistic regression classifier. The instance labeled **A** is far from the decision boundary; it would be predicted to belong to the positive class with a high probability. The instance labeled **B** would still be predicted to belong to the positive class, but the probability would be lower as the instance is closer to the decision boundary. Finally, the instance labeled **C** would be predicted to belong to the positive class with a low probability; even a small change to the training data could change the class that is predicted. The most confident predictions are for the instances that are farthest from the decision boundary. We can estimate the confidence of the prediction using its **functional margin**. The functional margin of the training set is given by the following equations:

$$funct = \min\ y_i f\left(x_i\right)$$

$$f\left(x\right) = \left\langle w, x \right\rangle + b$$

In the preceding formulae $y_i$ is the true class of the instance. The functional margin is large for instance **A** and small for instance **C**. If **C** were misclassified, the functional margin would be negative. The instances for which the functional margin is equal to one are called **support vectors**. These instances alone are sufficient to define the decision boundary; the other instances are not required to predict the class of a test instance. Related to the functional margin is the **geometric margin**, or the maximum width of the band that separates the support vectors. The geometric margin is equal to the normalized functional margin. It is necessary to normalize the functional margins as they can be scaled by using $w$, which is problematic for training. When $w$ is a unit vector, the geometric margin is equal to the functional vector. We can now formalize our definition of the best decision boundary as having the greatest geometric margin. The model parameters that maximize the geometric margin can be solved through the following constrained optimization problem:

$$\min \frac{1}{n} \langle w, w \rangle$$

$$\text{subject to}: y_i \left( \langle w, x_i \rangle + b \right) \geq 1$$

A useful property of support vector machines is that this optimization problem is convex; it has a single local minimum that is also the global minimum. While the proof is beyond the scope of this chapter, the previous optimization problem can be written using the dual form of the model to accommodate kernels as follows:

$$W(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_i K(x_i, x_j)$$

$$\text{subject to}: \sum_{i=1}^{n} y_i \alpha_i = 0$$

$$\text{subject to}: \alpha_i \geq 0$$

Finding the parameters that maximize the geometric margin subject to the constraints that all of the positive instances have functional margins of at least 1 and all of the negative instances have functional margins of at most -1 is a quadratic programming problem. This problem is commonly solved using an algorithm called **Sequential Minimal Optimization** (**SMO**). The SMO algorithm breaks the optimization problem down into a series of the smallest possible subproblems, which are then solved analytically.

# Classifying characters in scikit-learn

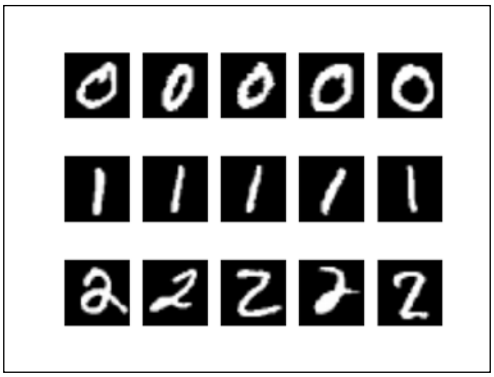
Let's apply support vector machines to a classification problem. In recent years, support vector machines have been used successfully in the task of character recognition. Given an image, the classifier must predict the character that is depicted. Character recognition is a component of many optical character-recognition systems. Even small images require high-dimensional representations when raw pixel intensities are used as features. If the classes are linearly inseparable and must be mapped to a higher-dimensional feature space, the dimensions of the feature space can become even larger. Fortunately, SVMs are suited to working with such data efficiently. First, we will use scikit-learn to train a support vector machine to recognize handwritten digits. Then, we will work on a more challenging problem: recognizing alphanumeric characters in photographs.

# Classifying handwritten digits

The Mixed National Institute of Standards and Technology database is a collection of 70,000 images of handwritten digits. The digits were sampled from documents written by employees of the US Census Bureau and American high school students. The images are grayscale and 28 x 28 pixels in dimension. Let's inspect some of the images using the following script:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.datasets import fetch_mldata
>>> import matplotlib.cm as cm

>>> digits = fetch_mldata('MNIST original', data_home='data/mnist').
data
>>> counter = 1
>>> for i in range(1, 4):
>>>     for j in range(1, 6):
>>>         plt.subplot(3, 5, counter)
>>>         plt.imshow(digits[(i - 1) * 8000 + j].reshape((28, 28)),
cmap=cm.Greys_r)
>>>         plt.axis('off')
>>>         counter += 1
>>> plt.show()
```

First, we load the data. scikit-learn provides the `fetch_mldata` convenience function to download the data set if it is not found on disk, and read it into an object. Then, we create a subplot for five instances for the digits zero, one, and two. The script produces the following figure:



The MNIST data set is partitioned into a training set of 60,000 images and test set of 10,000 images. The dataset is commonly used to evaluate a variety of machine learning models; it is popular because little preprocessing is required. Let's use scikit-learn to build a classifier that can predict the digit depicted in an image.

First, we import the necessary classes:

```
from sklearn.datasets import fetch_mldata
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import scale
from sklearn.cross_validation import train_test_split
from sklearn.svm import SVC
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import classification_report
```

The script will fork additional processes during grid search, which requires execution from a `__main__` block.

```
if __name__ == '__main__':
    data = fetch_mldata('MNIST original', data_home='data/mnist')
    X, y = data.data, data.target
    X = X/255.0*2 - 1
```

Next, we load the data using the `fetch_mldata` convenience function. We scale the features and center each feature around the origin. We then split the preprocessed data into training and test sets using the following line of code:

```
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

Next, we instantiate an `SVC`, or support vector classifier, object. This object exposes an API like that of scikit-learn's other estimators; the classifier is trained using the `fit` method, and predictions are made using the `predict` method. If you consult the documentation for `SVC`, you will find that the estimator requires more hyperparameters than most of the other estimators we discussed. It is common for more powerful estimators to require more hyperparameters. The most interesting hyperparameters for `SVC` are set by the `kernel`, `gamma`, and `C` keyword arguments. The `kernel` keyword argument specifies the kernel to be used. scikit-learn provides implementations of the linear, polynomial, sigmoid, and radial basis function kernels. The `degree` keyword argument should also be set when the polynomial kernel is used. `C` controls regularization; it is similar to the lambda hyperparameter we used for logistic regression. The keyword argument `gamma` is the kernel coefficient for the sigmoid, polynomial, and RBF kernels. Setting these hyperparameters can be challenging, so we tune them by grid searching with the following code.

```
pipeline = Pipeline([
    ('clf', SVC(kernel='rbf', gamma=0.01, C=100))
])
print X_train.shape
parameters = {
    'clf__gamma': (0.01, 0.03, 0.1, 0.3, 1),
    'clf__C': (0.1, 0.3, 1, 3, 10, 30),
}
grid_search = GridSearchCV(pipeline, parameters, n_jobs=2,
verbose=1, scoring='accuracy')
grid_search.fit(X_train[:10000], y_train[:10000])
print 'Best score: %0.3f' % grid_search.best_score_
print 'Best parameters set:'
best_parameters = grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print '\t%s: %r' % (param_name, best_parameters[param_name])
predictions = grid_search.predict(X_test)
print classification_report(y_test, predictions)
```

The following is the output of the preceding script:

```
Fitting 3 folds for each of 30 candidates, totalling 90 fits
[Parallel(n_jobs=2)]: Done   1 jobs       | elapsed:  7.7min
[Parallel(n_jobs=2)]: Done  50 jobs       | elapsed: 201.2min
[Parallel(n_jobs=2)]: Done  88 out of  90 | elapsed: 304.8min
remaining:  6.9min
[Parallel(n_jobs=2)]: Done  90 out of  90 | elapsed: 309.2min finished
Best score: 0.966
Best parameters set:
   clf__C: 3
   clf__gamma: 0.01
             precision    recall  f1-score   support

        0.0       0.98      0.99      0.99      1758
        1.0       0.98      0.99      0.98      1968
        2.0       0.95      0.97      0.96      1727
        3.0       0.97      0.95      0.96      1803
        4.0       0.97      0.98      0.97      1714
        5.0       0.96      0.96      0.96      1535
        6.0       0.98      0.98      0.98      1758
        7.0       0.97      0.96      0.97      1840
        8.0       0.95      0.96      0.96      1668
        9.0       0.96      0.95      0.96      1729

avg / total       0.97      0.97      0.97     17500
```

The best model has an average F1 score of 0.97; this score can be increased further by training on more than the first ten thousand instances.

# Classifying characters in natural images

Now let's try a more challenging problem. We will classify alphanumeric characters in natural images. The Chars74K dataset, collected by T. E. de Campos, B. R. Babu, and M. Varma for *Character Recognition in Natural Images*, contains more than 74,000 images of the digits zero through to nine and the characters for both cases of the English alphabet. The following are three examples of images of the lowercase letter `z`. Chars74K can be downloaded from `http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/`.

Several types of images comprise the collection. We will use 7,705 images of characters that were extracted from photographs of street scenes taken in Bangalore, India. In contrast to MNIST, the images in this portion of Chars74K depict the characters in a variety of fonts, colors, and perturbations. After expanding the archive, we will use the files in the `English/Img/GoodImg/Bmp/` directory. First we will import the necessary classes.

```
import os
import numpy as np
from sklearn.svm import SVC
from sklearn.cross_validation import train_test_split
from sklearn.metrics import classification_report
import Image
```

Next we will define a function that resizes images using the Python Image Library:

```
def resize_and_crop(image, size):
    img_ratio = image.size[0] / float(image.size[1])
    ratio = size[0] / float(size[1])
    if ratio > img_ratio:
        image = image.resize((size[0], size[0] * image.size[1] /
image.size[0]), Image.ANTIALIAS)
        image = image.crop((0, 0, 30, 30))
    elif ratio < img_ratio:
        image = image.resize((size[1] * image.size[0] / image.size[1],
size[1]), Image.ANTIALIAS)
        image = image.crop((0, 0, 30, 30))
    else:
        image = image.resize((size[0], size[1]), Image.ANTIALIAS)
    return image
```

Then we load will the images for each of the 62 classes and convert them to grayscale. Unlike MNIST, the images of Chars74K do not have consistent dimensions, so we will resize them to 30 pixels on a side using the resize_and_crop function we defined. Finally, we will convert the processed images to a NumPy array:

```
X = []
y = []

for path, subdirs, files in os.walk('data/English/Img/GoodImg/Bmp/'):
    for filename in files:
        f = os.path.join(path, filename)
        img = Image.open(f).convert('L') # convert to grayscale
        img_resized = resize_and_crop(img, (30, 30))
        img_resized = np.asarray(img_resized.getdata(), dtype=np.
float64) \
            .reshape((img_resized.size[1] * img_resized.size[0], 1))
        target = filename[3:filename.index('-')]
        X.append(img_resized)
        y.append(target)

X = np.array(X)
X = X.reshape(X.shape[:2])

We will then train a support vector classifier with a polynomial
kernel.classifier = SVC(verbose=0, kernel='poly', degree=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_
state=1)
classifier.fit(X_train, y_train)
predictions = classifier.predict(X_test)
print classification_report(y_test, predictions)
```

The preceding script produces the following output:

```
             precision    recall  f1-score   support

        001       0.24      0.22      0.23        23
        002       0.24      0.45      0.32        20
       ...
        061       0.33      0.15      0.21        13
        062       0.08      0.25      0.12         8

avg / total       0.41      0.34      0.36      1927
```

It is apparent that this is a more challenging task than classifying digits in MNIST. The appearances of the characters vary more widely, the characters are perturbed more since the images were sampled from photographs rather than scanned documents. Furthermore, there are far fewer training instances for each class in Chars74K than there are in MNIST. The performance of the classifier could be improved by adding training data, preprocessing the images differently, or using more sophisticated feature representations.

# Summary

In this chapter, we discussed the support vector machine—a powerful model that can mitigate some of the limitations of perceptrons. The perceptron can be used effectively for linearly separable classification problems, but it cannot express more complex decision boundaries without expanding the feature space to higher dimensions. Unfortunately, this expansion is prone to computation and generalization problems. Support vector machines redress the first problem using kernels, which avoid explicitly computing the feature mapping. They redress the second problem by maximizing the margin between the decision boundary and the nearest instances. In the next chapter, we will discuss models called artificial neural networks, which, like support vector machines, extend the perceptron to overcome its limitations.

# 10
# From the Perceptron to Artificial Neural Networks

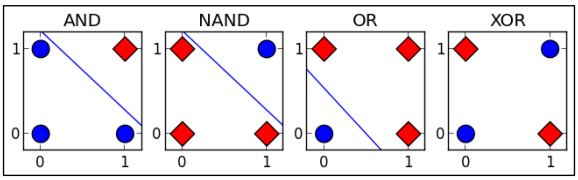
In *Chapter 8*, *The Perceptron*, we introduced the perceptron, which is a linear model for binary classification. You learned that the perceptron is not a universal function approximator; its decision boundary must be a hyperplane. In the previous chapter we introduced the support vector machine, which redresses some of the perceptron's limitations by using kernels to efficiently map the feature representations to a higher dimensional space in which the instances are linearly separable. In this chapter, we will discuss **artificial neural networks**, which are powerful nonlinear models for classification and regression that use a different strategy to overcome the perceptron's limitations.

If the perceptron is analogous to a neuron, an artificial neural network, or **neural net**, is analogous to a brain. As billions of neurons with trillions of synapses comprise a human brain, an artificial neural network is a directed graph of perceptrons or other artificial neurons. The graph's edges are weighted; these weights are the parameters of the model that must be learned.

Entire books describe individual aspects of artificial neural networks; this chapter will provide an overview of their structure and training. At the time of writing, some artificial neural networks have been developed for scikit-learn, but they are not available in Version 0.15.2. Readers can follow the examples in this chapter by checking out a fork of scikit-learn 0.15.1 that includes the neural network module. The implementations in this fork are likely to be merged into future versions of scikit-learn without any changes to the API described in this chapter.

# Nonlinear decision boundaries

Recall from *Chapter 8*, *The Perceptron*, that while some Boolean functions such as AND, OR, and NAND can be approximated by the perceptron, the linearly inseparable function XOR cannot, as shown in the following plots:




Let's review XOR in more detail to develop an intuition for the power of artificial neural networks. In contrast to AND, which outputs 1 when both of its inputs are equal to 1, and OR, which outputs 1 when at least one of the inputs are equal to 1, the output of XOR is 1 when exactly one of its inputs are equal to 1. We could view XOR as outputting 1 when two conditions are true. The first condition is that at least one of the inputs must be equal to 1; this is the same condition that OR tests. The second condition is that not both of the inputs are equal to 1; NAND tests this condition. We can produce the same output as XOR by processing the input with both OR and NAND and then verifying that the outputs of both functions are equal to 1 using AND. That is, the functions OR, NAND, and AND can be composed to produce the same output as XOR.

The following tables provide the truth tables for XOR, OR, AND, and NAND for the inputs *A* and *B*. From these tables we can verify that inputting the output of OR and NAND to AND produces the same output as inputting *A* and *B* to XOR:

| A | B | A AND B | A NAND B | A OR B | A XOR B |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 |

| A | B | A OR B | A NAND B | (A OR B) AND (A NAND B) |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |

Instead of trying to represent XOR with a single perceptron, we will build an artificial neural network from multiple artificial neurons that each approximate a linear function. Each instance's feature representation will be input to two neurons; one neuron will represent NAND and the other will represent OR. The output of these neurons will be received by a third neuron that represents AND to test whether both of XOR's conditions are true.

# Feedforward and feedback artificial neural networks

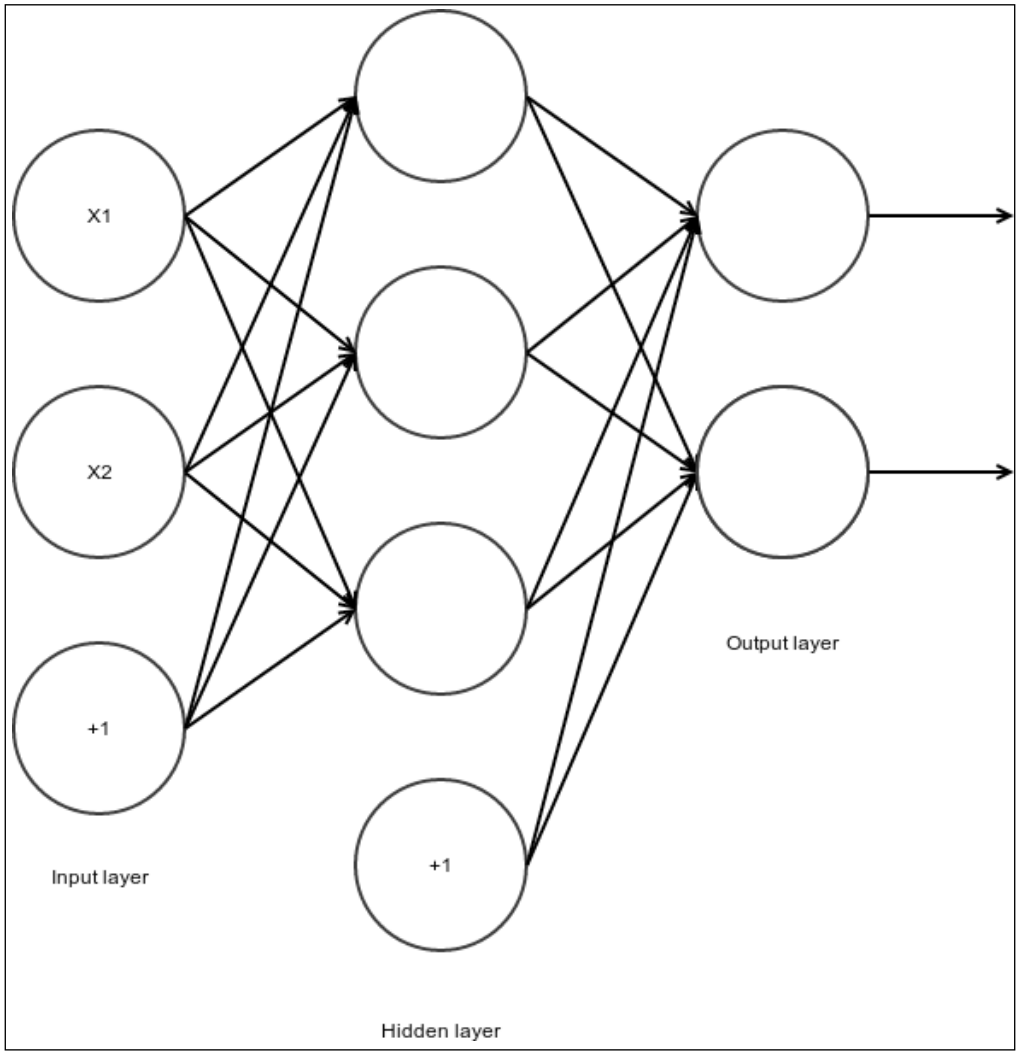
Artificial neural networks are described by three components. The first is the model's **architecture**, or topology, which describes the layers of neurons and structure of the connections between them. The second component is the activation function used by the artificial neurons. The third component is the learning algorithm that finds the optimal values of the weights.

There are two main types of artificial neural networks. **Feedforward neural networks** are the most common type of neural net, and are defined by their directed acyclic graphs. Signals only travel in one direction—towards the output layer—in feedforward neural networks. Conversely, **feedback neural networks**, or recurrent neural networks, do contain cycles. The feedback cycles can represent an internal state for the network that can cause the network's behavior to change over time based on its input. Feedforward neural networks are commonly used to learn a function to map an input to an output. The temporal behavior of feedback neural networks makes them suitable for processing sequences of inputs. Because feedback neural networks are not implemented in scikit-learn, we will limit our discussion to only feedforward neural networks.

# Multilayer perceptrons

The **multilayer perceptron** (**MLP**) is the one of the most commonly used artificial neural networks. The name is a slight misnomer; a multilayer perceptron is not a single perceptron with multiple layers, but rather multiple layers of artificial neurons that can be perceptrons. The layers of the MLP form a directed, acyclic graph. Generally, each layer is fully connected to the subsequent layer; the output of each artificial neuron in a layer is an input to every artificial neuron in the next layer towards the output. MLPs have three or more layers of artificial neurons.

The **input layer** consists of simple input neurons. The input neurons are connected to at least one **hidden layer** of artificial neurons. The hidden layer represents latent variables; the input and output of this layer cannot be observed in the training data. Finally, the last hidden layer is connected to an **output layer**. The following diagram depicts the architecture of a multilayer perceptron with three layers. The neurons labeled **+1** are bias neurons and are not depicted in most architecture diagrams.



The artificial neurons, or **units**, in the hidden layer commonly use nonlinear activation functions such as the hyperbolic tangent function and the logistic function, which are given by the following equations:

$$f(x) = \tanh(x)$$

$$f(x) = \frac{1}{1+e^{-x}}$$

As with other supervised models, our goal is to find the values of the weights that minimize the value of a cost function. The mean squared error cost function is commonly used with multilayer perceptrons. It is given by the following equation, where $m$ is the number of training instances:

$$MSE = \frac{1}{m}\sum_{i=1} m\left(y_i - f\left(x_i\right)\right)^2$$

# Minimizing the cost function

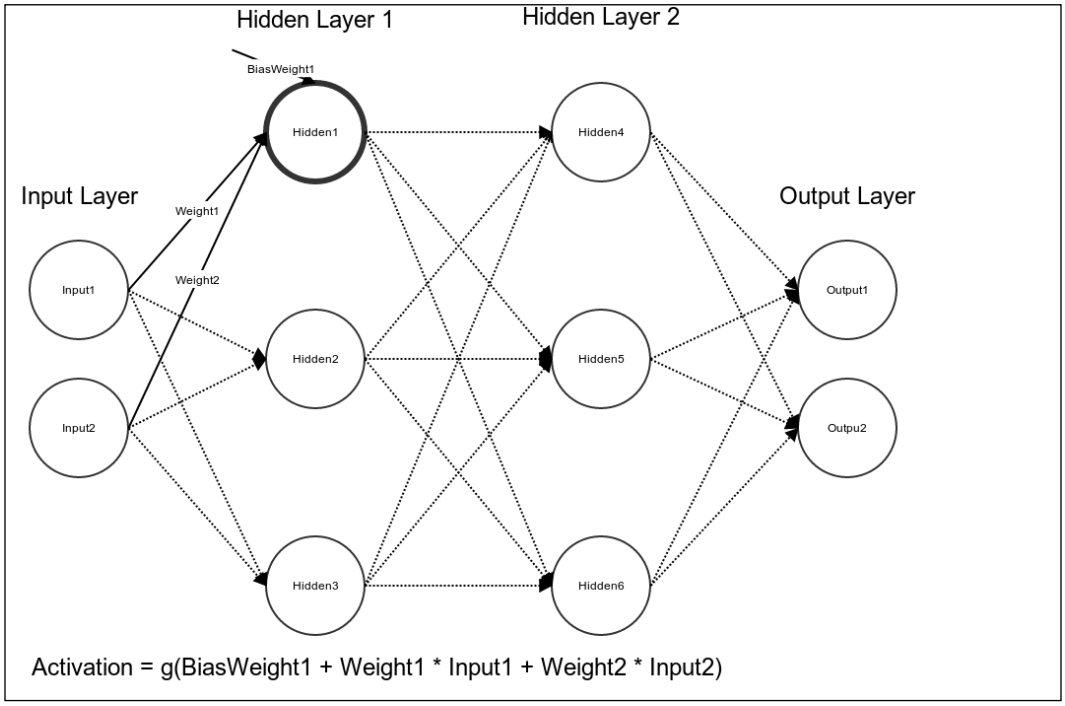
The **backpropagation** algorithm is commonly used in conjunction with an optimization algorithm such as gradient descent to minimize the value of the cost function. The algorithm takes its name from a portmanteau of *backward propagation*, and refers to the direction in which errors flow through the layers of the network. Backpropagation can theoretically be used to train a feedforward network with any number of hidden units arranged in any number of layers, though computational power constrains this capability.

Backpropagation is similar to gradient descent in that it uses the gradient of the cost function to update the values of the model parameters. Unlike the linear models we have previously seen, neural nets contain hidden units that represent latent variables; we can't tell what the hidden units should do from the training data. If we do not know what the hidden units should do, we cannot calculate their errors and we cannot calculate the gradient of cost function with respect to their weights. A naive solution to overcome this is to randomly perturb the weights for the hidden units. If a random change to one of the weights decreases the value of the cost function, we save the change and randomly change the value of another weight. An obvious problem with this solution is its prohibitive computational cost. Backpropagation provides a more efficient solution.

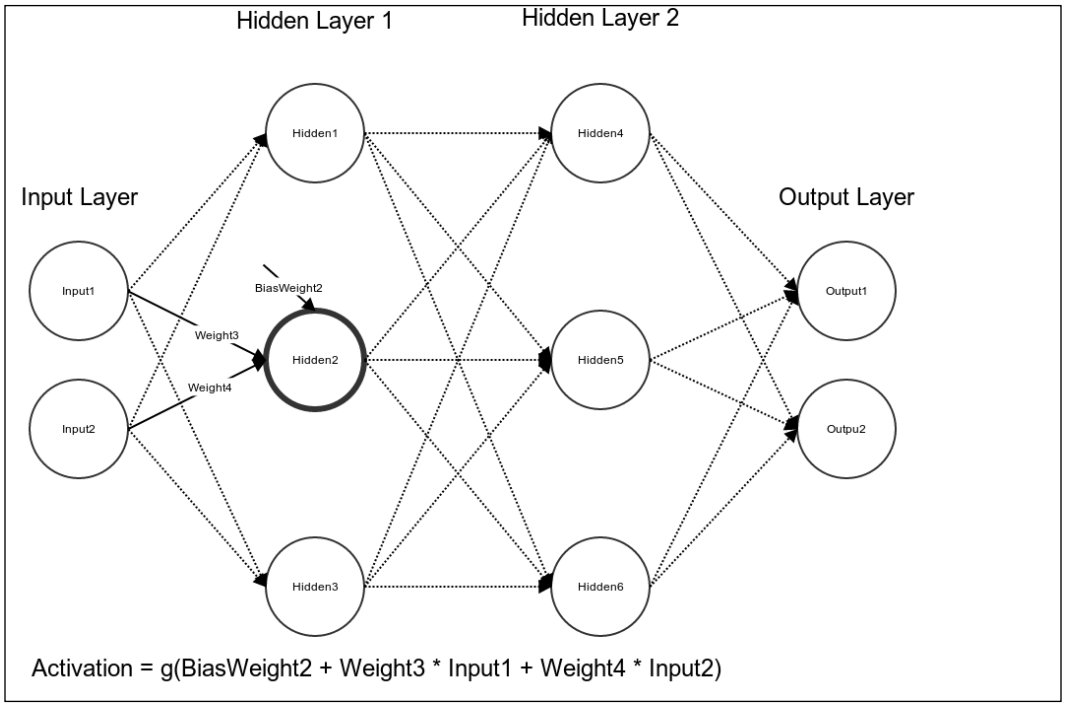
We will step through training a feedforward neural network using backpropagation. This network has two input units, two hidden layers that both have three hidden units, and two output units. The input units are both fully connected to the first hidden layer's units, called `Hidden1`, `Hidden2`, and `Hidden3`. The edges connecting the units are initialized to small random weights.

# Forward propagation

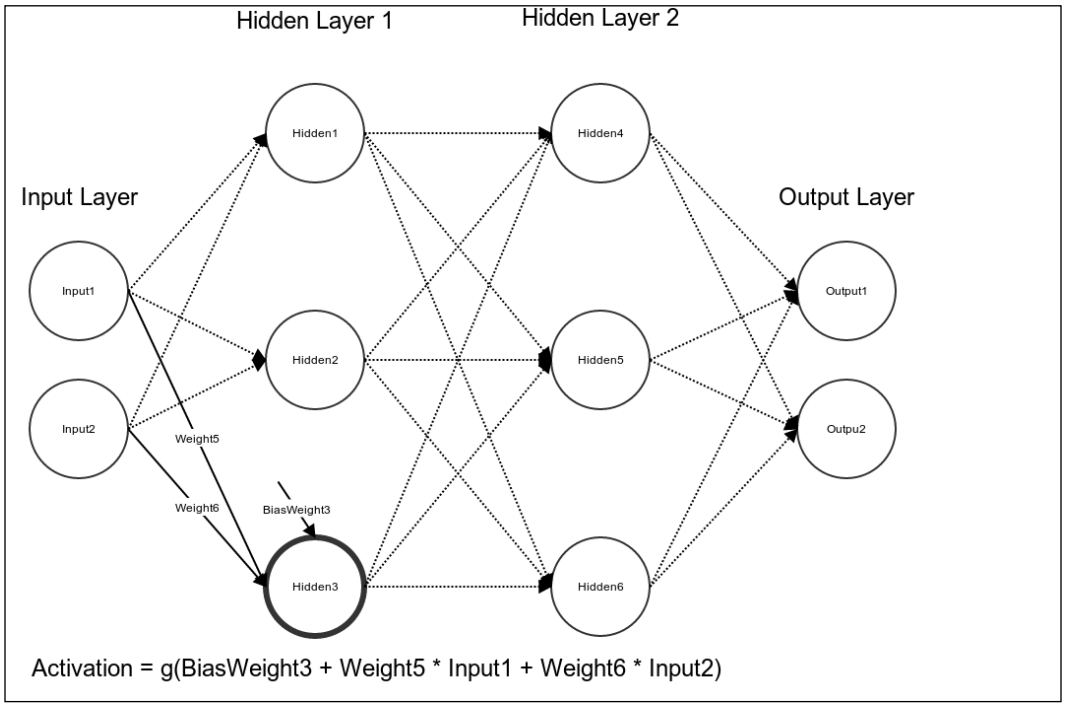
During the forward propagation stage, the features are input to the network and fed through the subsequent layers to produce the output activations. First, we compute the activation for the unit `Hidden1`. We find the weighted sum of input to `Hidden1`, and then process the sum with the activation function. Note that `Hidden1` receives a constant input from a bias unit that is not depicted in the diagram in addition to the inputs from the input units. In the following diagram, $g(x)$ is the activation function:



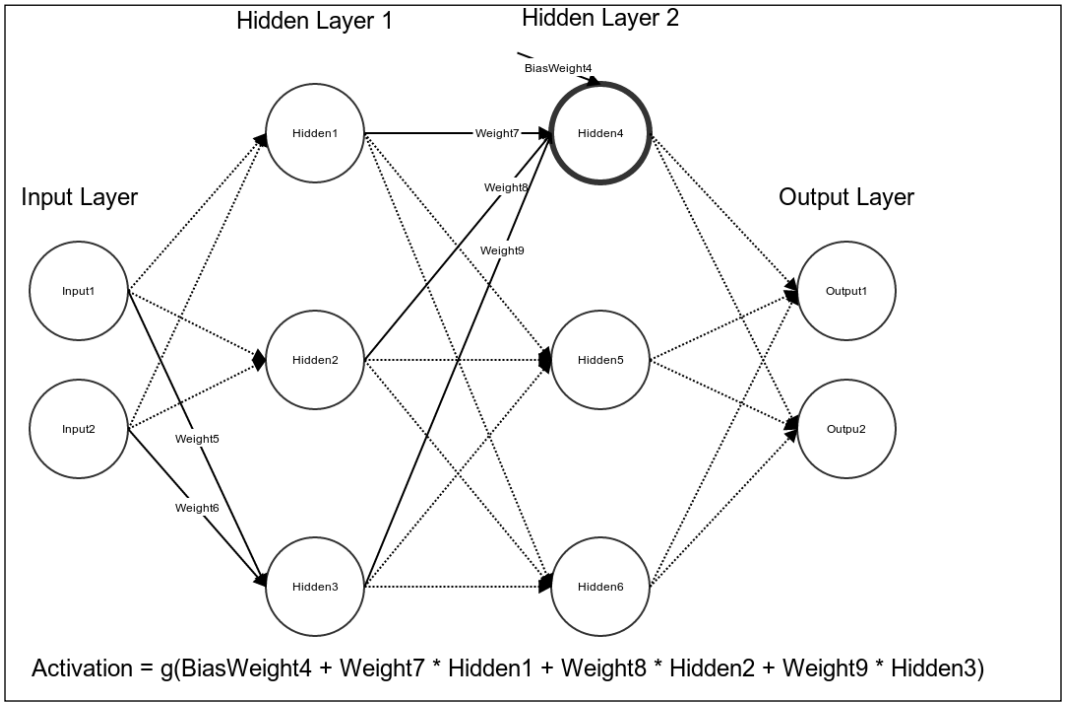
Next, we compute the activation for the second hidden unit. Like the first hidden unit, it receives weighted inputs from both of the input units and a constant input from a bias unit. We then process the weighted sum of the inputs, or **preactivation**, with the activation function as shown in the following figure:



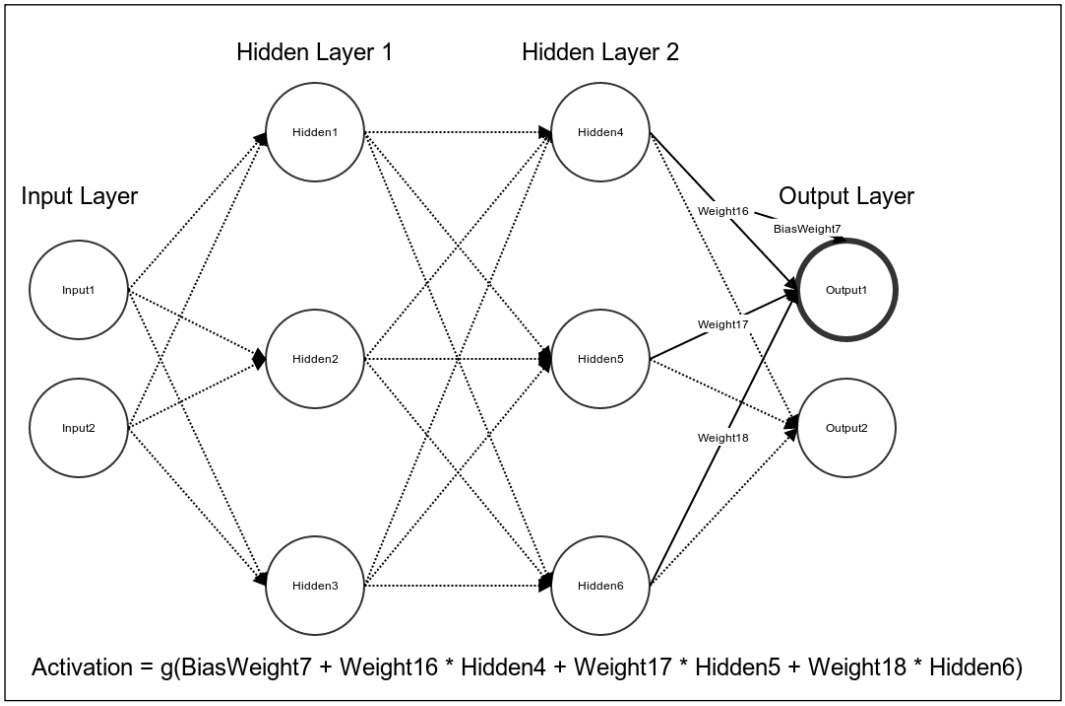
We then compute the activation for `Hidden3` in the same manner:



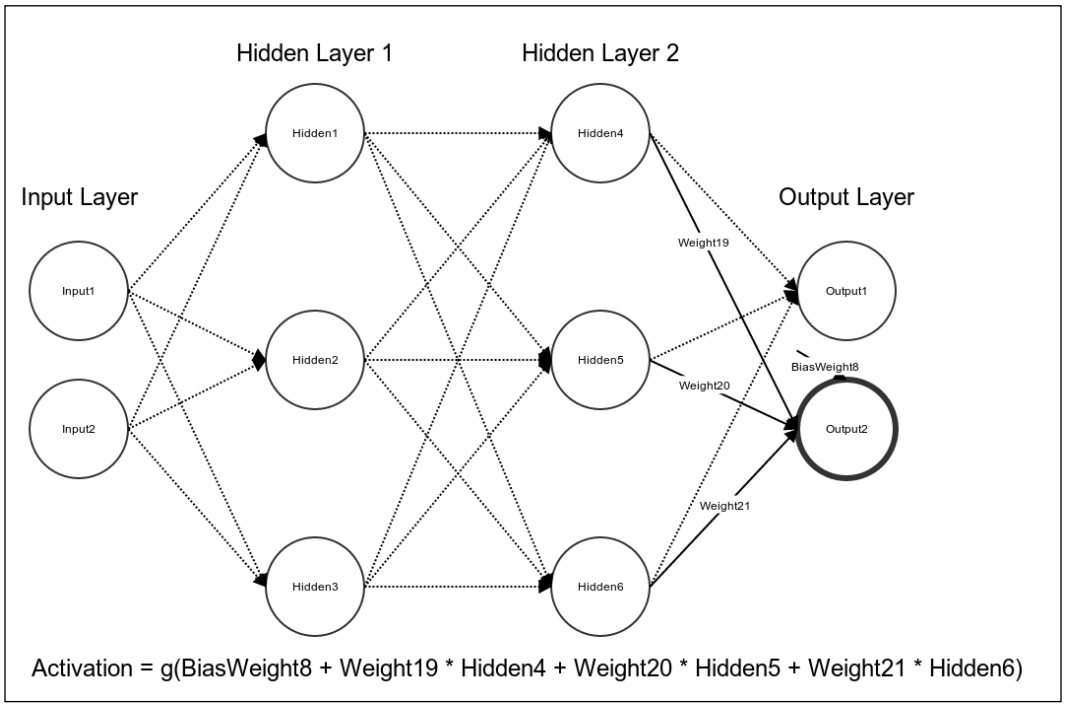
Having computed the activations of all of the hidden units in the first layer, we proceed to the second hidden layer. In this network, the first hidden layer is fully connected to the second hidden layer. Similar to the units in the first hidden layer, the units in the second hidden layer receive a constant input from bias units that are not depicted in the diagram. We proceed to compute the activation of `Hidden4`:



We next compute the activations of `Hidden5` and `Hidden6`. Having computed the activations of all of the hidden units in the second hidden layer, we proceed to the output layer in the following figure. The activation of `Output1` is the weighted sum of the second hidden layer's activations processed through an activation function. Similar to the hidden units, the output units both receive a constant input from a bias unit:



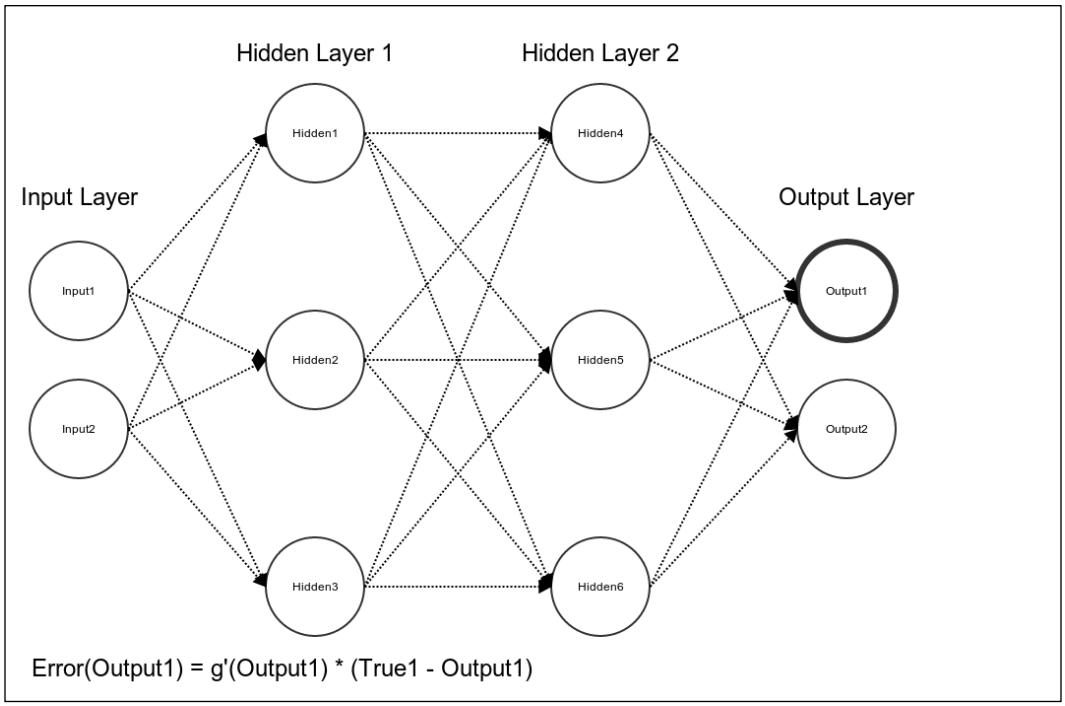
We calculate the activation of `Output2` in the same manner:



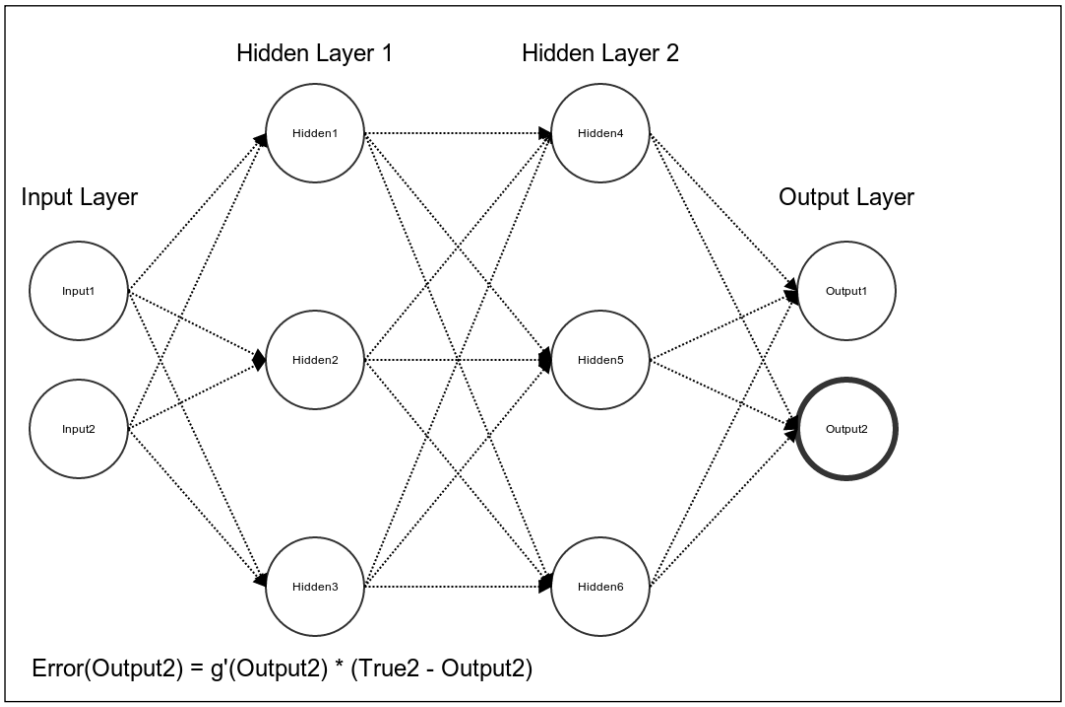

We computed the activations of all of the units in the network, and we have now completed forward propagation. The network is not likely to approximate the true function well using the initial random values of the weights. We must now update the values of the weights so that the network can better approximate our function.

# Backpropagation

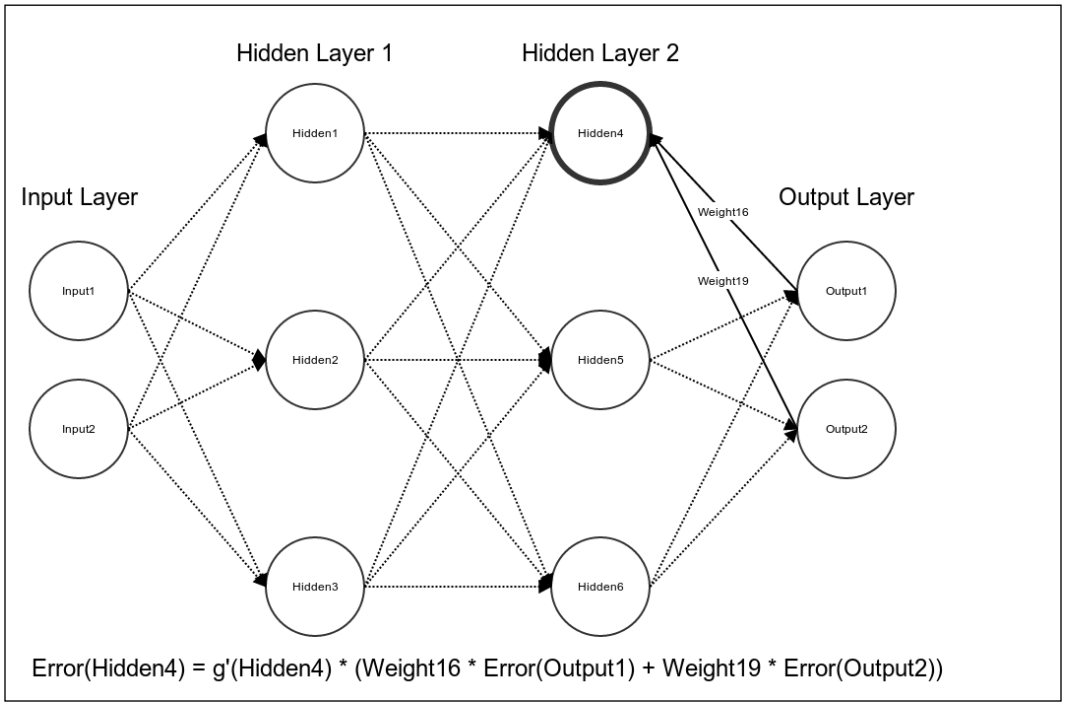
We can calculate the error of the network only at the output units. The hidden units represent latent variables; we cannot observe their true values in the training data and thus, we have nothing to compute their error against. In order to update their weights, we must propagate the network's errors backwards through its layers. We will begin with `Output1`. Its error is equal to the difference between the true and predicted outputs, multiplied by the partial derivative of the unit's activation:



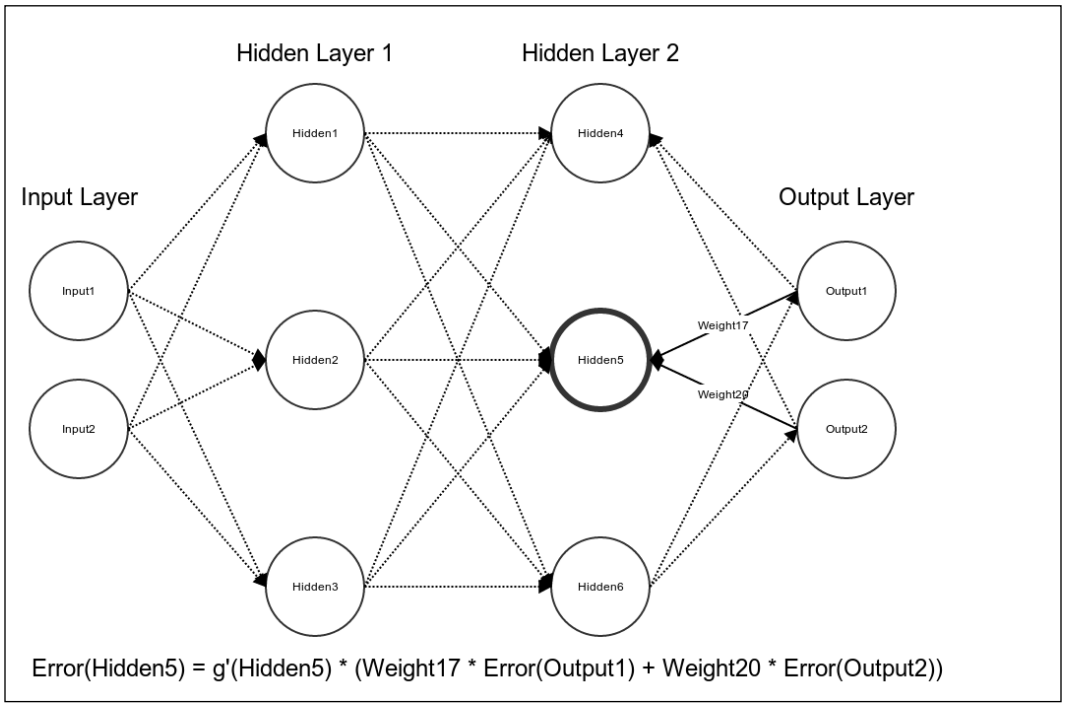
We then calculate the error of the second output unit:



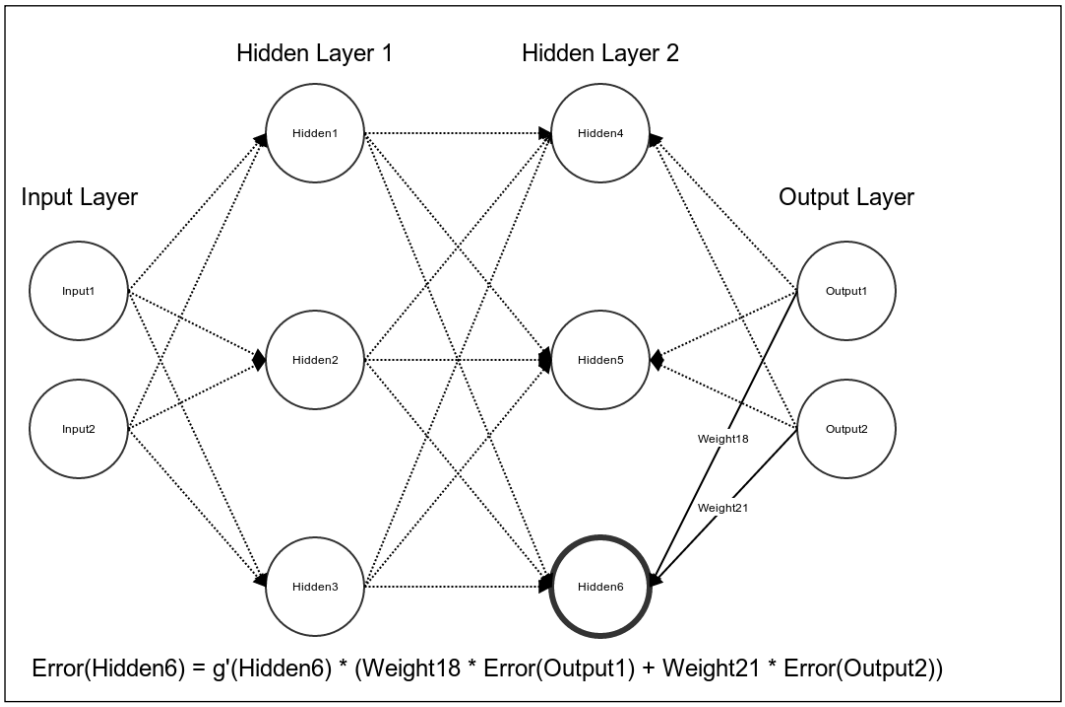
We computed the errors of the output layer. We can now propagate these errors backwards to the second hidden layer. First, we will compute the error of hidden unit `Hidden4`. We multiply the error of `Output1` by the value of the weight connecting `Hidden4` and `Output1`. We similarly weigh the error of `Output2`. We then add these errors and calculate the product of their sum and the partial derivative of `Hidden4`:



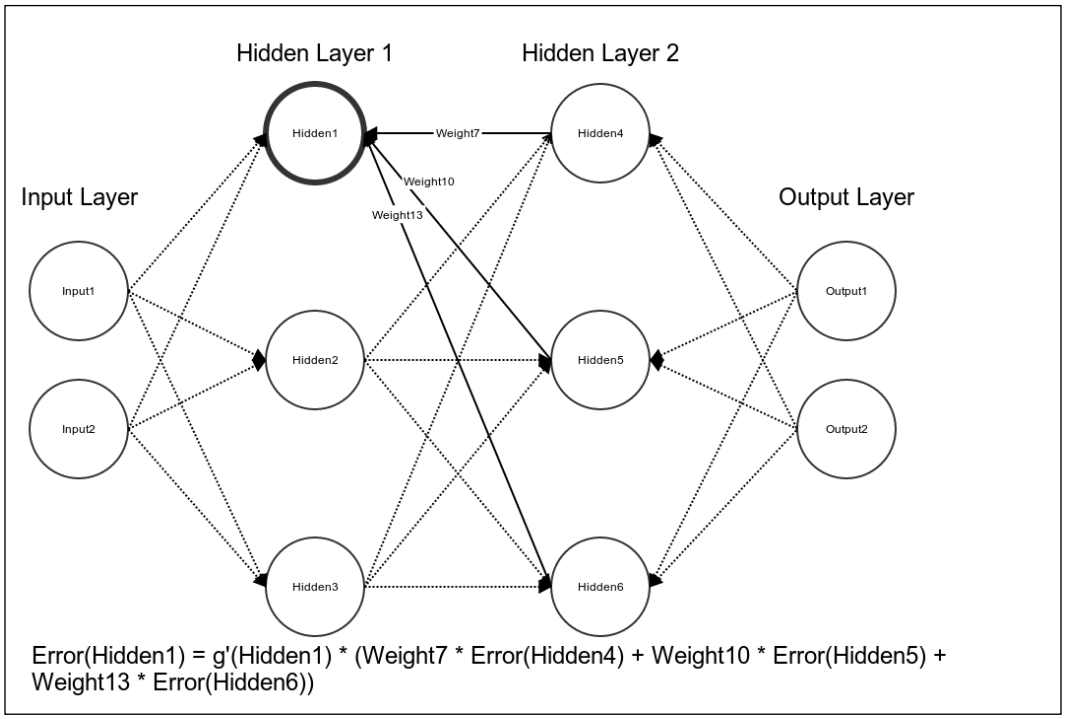
We similarly compute the errors of `Hidden5`:



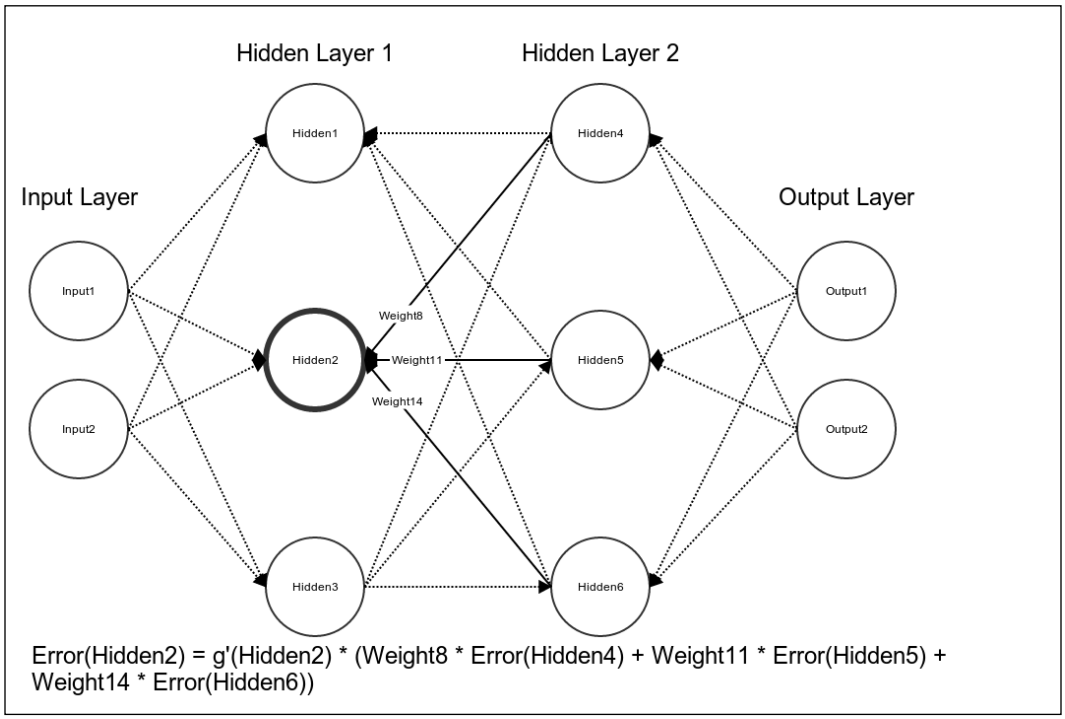
We then compute the `Hidden6` error in the following figure:



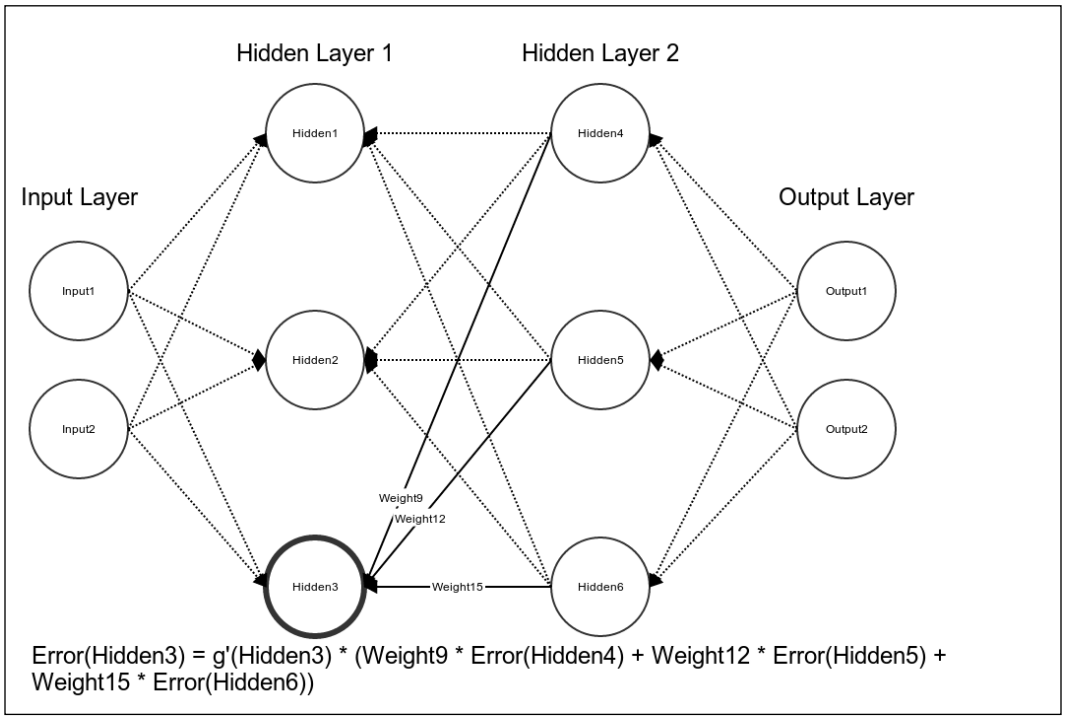
We calculated the error of the second hidden layer with respect to the output layer. Next, we will continue to propagate the errors backwards towards the input layer. The error of the hidden unit `Hidden1` is the product of its partial derivative and the weighted sums of the errors in the second hidden layer:



We similarly compute the error for hidden unit `Hidden2`:

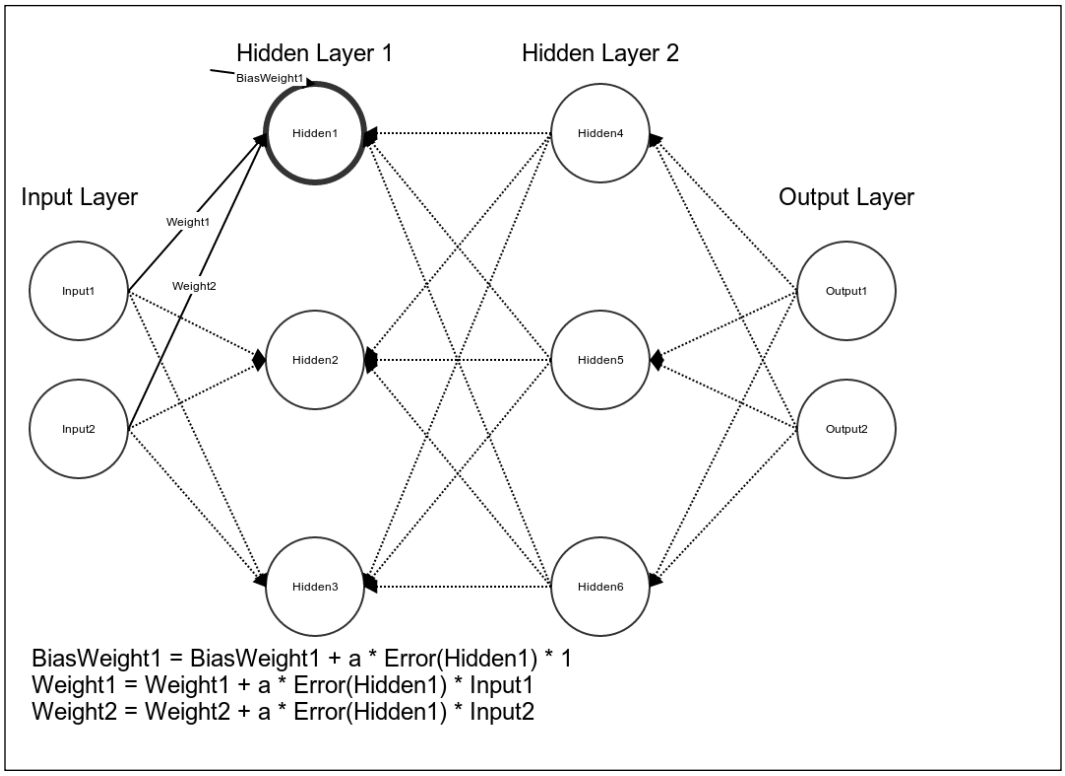


We similarly compute the error for `Hidden3`:

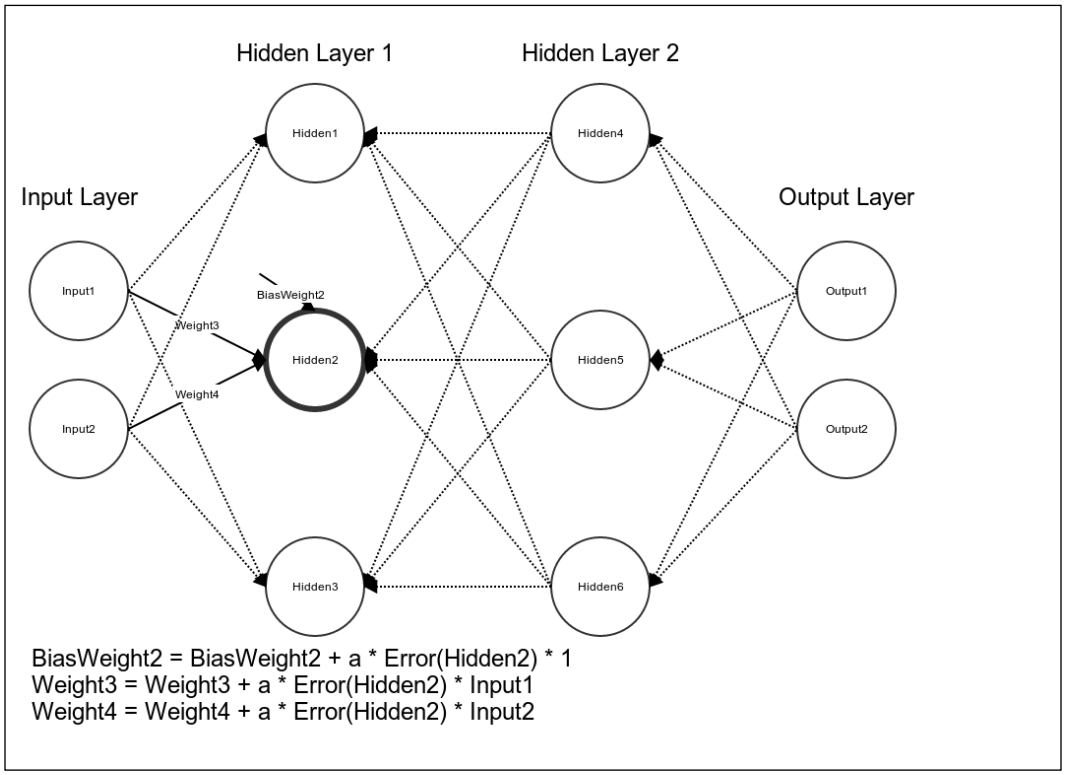



We computed the errors of the first hidden layer. We can now use these errors to update the values of the weights. We will first update the weights for the edges connecting the input units to `Hidden1` as well as the weight for the edge connecting the bias unit to `Hidden1`. We will increment the value of the weight connecting `Input1` and `Hidden1` by the product of the learning rate, error of `Hidden1`, and the value of `Input1`.

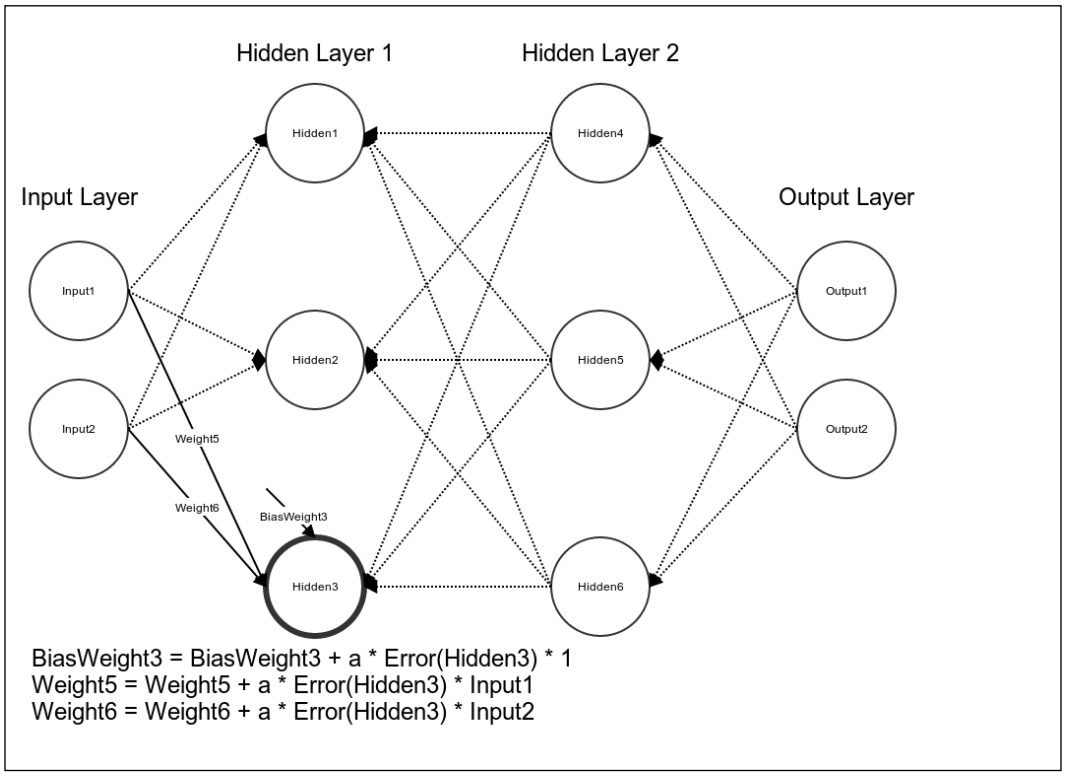
We will similarly increment the value of `Weight2` by the product of the learning rate, error of `Hidden1`, and the value of `Input2`. Finally, we will increment the value of the weight connecting the bias unit to `Hidden1` by the product of the learning rate, error of `Hidden1`, and one.



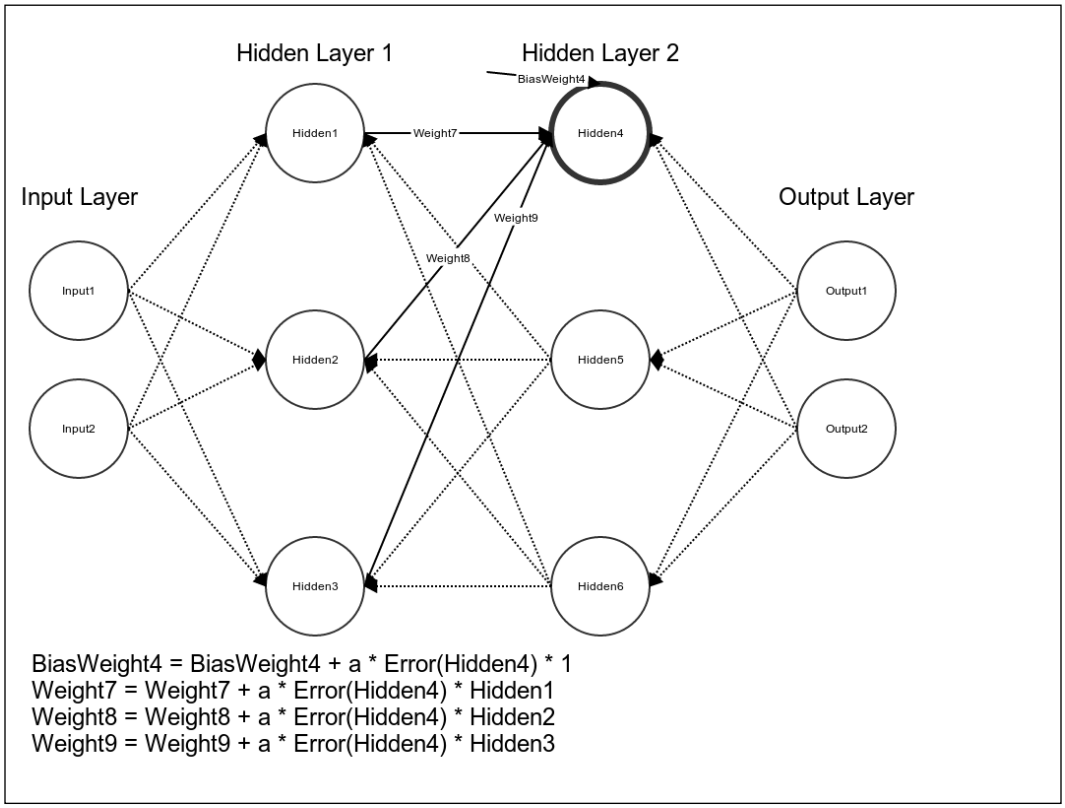
We will then update the values of the weights connecting hidden unit `Hidden2` to the input units and the bias unit using the same method:



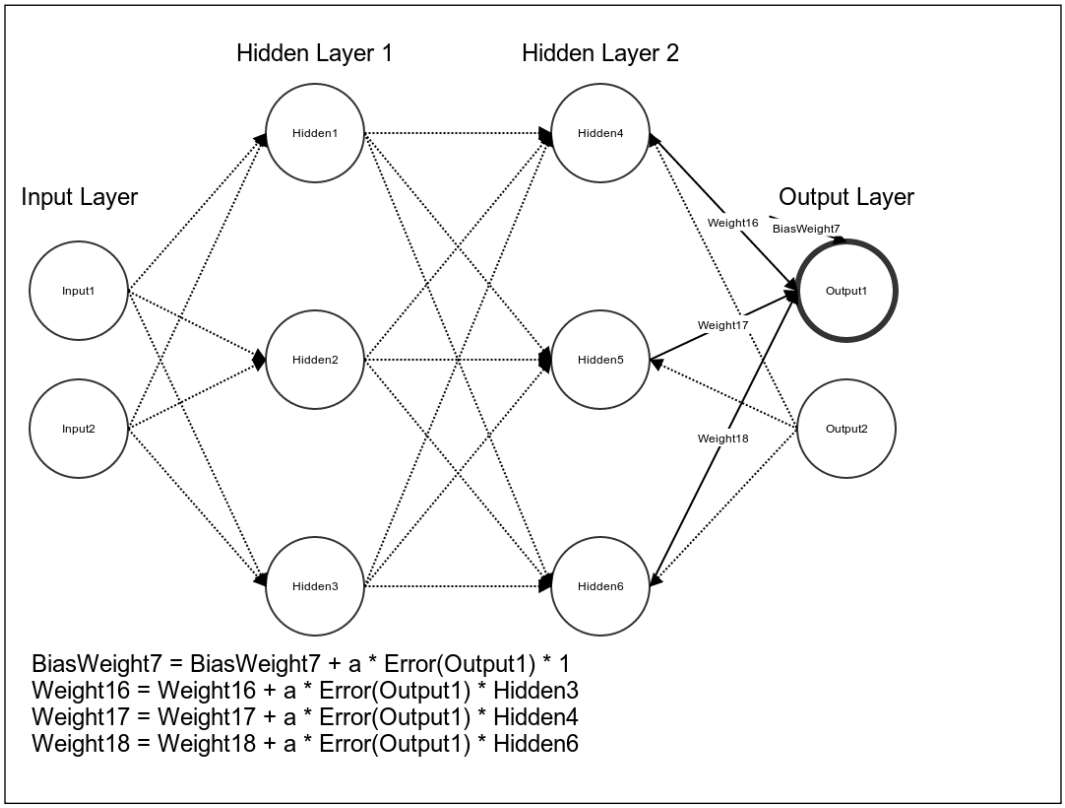
Next, we will update the values of the weights connecting the input layer to `Hidden3`:

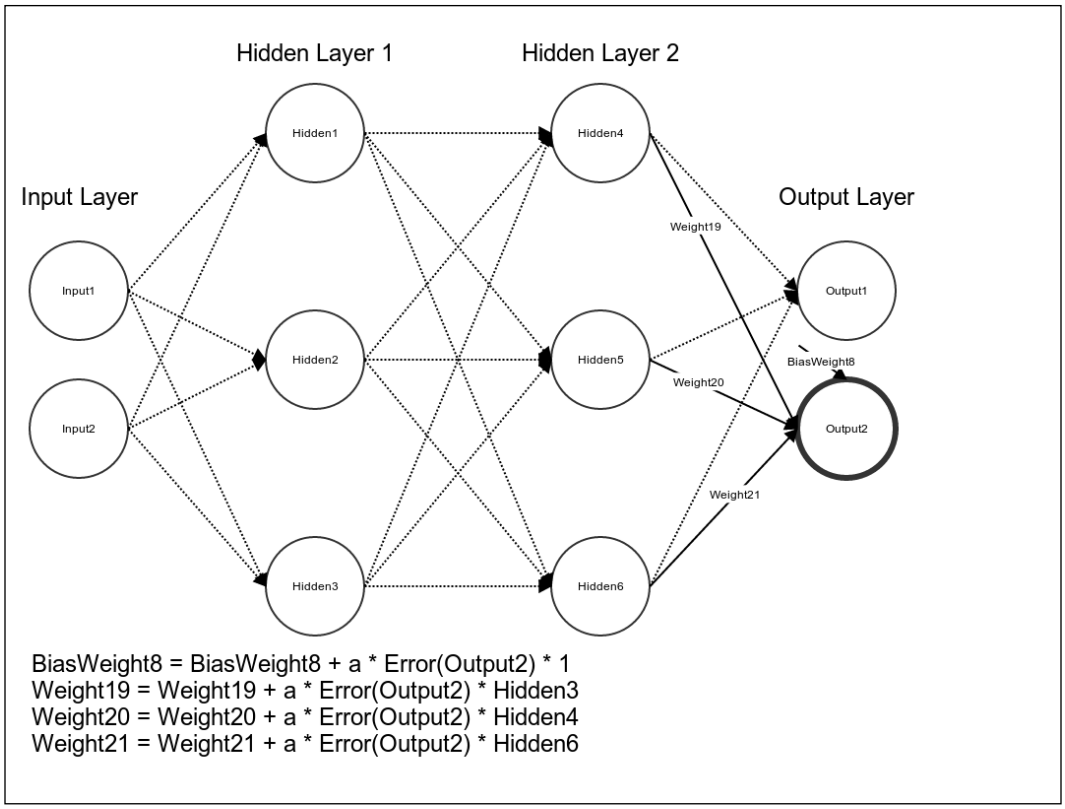


Since the values of the weights connecting the input layer to the first hidden layer is updated, we can continue to the weights connecting the first hidden layer to the second hidden layer. We will increment the value of `Weight7` by the product of the learning rate, error of `Hidden4`, and the output of `Hidden1`. We continue to similarly update the values of weights `Weight8` to `Weight15`:



The weights for `Hidden5` and `Hidden6` are updated in the same way. We updated the values of the weights connecting the two hidden layers. We can now update the values of the weights connecting the second hidden layer and the output layer. We increment the values of weights `W16` through `W21` using the same method that we used for the weights in the previous layers:






After incrementing the value of `Weight21` by the product of the learning rate, error of `Output2`, and the activation of `Hidden6`, we have finished updating the values of the weights for the network. We can now perform another forward pass using the new values of the weights; the value of the cost function produced using the updated weights should be smaller. We will repeat this process until the model converges or another stopping criterion is satisfied. Unlike the linear models we have discussed, backpropagation does not optimize a convex function. It is possible that backpropagation will converge on parameter values that specify a local, rather than global, minimum. In practice, local optima are frequently adequate for many applications.

# Approximating XOR with Multilayer perceptrons

Let's train a multilayer perceptron to approximate the XOR function. At the time of writing, multilayer perceptrons have been implemented as part of a 2014 Google Summer of Code project, but have not been merged or released. Subsequent versions of scikit-learn are likely to include this implementation of multilayer perceptrons without any changes to the API described in this section. In the interim, a fork of scikit-learn 0.15.1 that includes the multilayer perceptron implementation can be cloned from `https://github.com/IssamLaradji/scikit-learn.git`.

First, we will create a toy binary classification dataset that represents XOR and split it into training and testing sets:

```
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.neural_network import MultilayerPerceptronClassifier
>>> y = [0, 1, 1, 0] * 1000
>>> X = [[0, 0], [0, 1], [1, 0], [1, 1]] * 1000
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_
state=3)
```

Next we instantiate `MultilayerPerceptronClassifier`. We specify the architecture of the network through the `n_hidden` keyword argument, which takes a list of the number of hidden units in each hidden layer. We create a hidden layer with two units that use the logistic activation function. The `MultilayerPerceptronClassifier` class automatically creates two input units and one output unit. In multi-class problems the classifier will create one output unit for each of the possible classes.

Selecting an architecture is challenging. There are some rules of thumb to choose the numbers of hidden units and layers, but these tend to be supported only by anecdotal evidence. The optimal number of hidden units depends on the number of training instances, the noise in the training data, the complexity of the function that is being approximated, the hidden units' activation function, the learning algorithm, and the regularization employed. In practice, architectures can only be evaluated by comparing their performances through cross validation.

We train the network by calling the fit() method:

```
>>> clf = MultilayerPerceptronClassifier(n_hidden=[2],
>>>                                      activation='logistic',
>>>                                      algorithm='sgd',
>>>                                      random_state=3)
>>> clf.fit(X_train, y_train)
```

Finally, we print some predictions for manual inspection and evaluate the model's accuracy on the test set. The network perfectly approximates the XOR function on the test set:

```
>>> print 'Number of layers: %s. Number of outputs: %s' % (clf.n_
layers_, clf.n_outputs_)
>>> predictions = clf.predict(X_test)
>>> print 'Accuracy:', clf.score(X_test, y_test)
>>> for i, p in enumerate(predictions[:10]):
>>>     print 'True: %s, Predicted: %s' % (y_test[i], p)
Number of layers: 3. Number of outputs: 1
Accuracy: 1.0
True: 1, Predicted: 1
True: 1, Predicted: 1
True: 1, Predicted: 1
True: 0, Predicted: 0
True: 1, Predicted: 1
True: 0, Predicted: 0
True: 0, Predicted: 0
True: 1, Predicted: 1
True: 0, Predicted: 0
True: 1, Predicted: 1
```

# Classifying handwritten digits

In the previous chapter we used a support vector machine to classify the handwritten digits in the MNIST dataset. In this section we will classify the images using an artificial neural network:

```
from sklearn.datasets import load_digits
from sklearn.cross_validation import train_test_split, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network.multilayer_perceptron import
MultilayerPerceptronClassifier
```

First we use the `load_digits` convenience function to load the MNIST dataset. We will fork additional processes during cross validation, which requires execution from a `main`-protected block:

```
>>> if __name__ == '__main__':
>>>     digits = load_digits()
>>>     X = digits.data
>>>     y = digits.target
```

Scaling the features is particularly important for artificial neural networks and will help some learning algorithms to converge more quickly. Next, we create a `Pipeline` class that scales the data before fitting a `MultilayerPerceptronClassifier`. This network contains an input layer, a hidden layer with 150 units, a hidden layer with 100 units, and an output layer. We also increased the value of the regularization hyperparameter `alpha` argument. Finally, we print the accuracies of the three cross validation folds. The code is as follows:

```
>>>     pipeline = Pipeline([
>>>         ('ss', StandardScaler()),
>>>         ('mlp', MultilayerPerceptronClassifier(n_hidden=[150,
100], alpha=0.1))
>>>     ])
>>>     print cross_val_score(pipeline, X, y, n_jobs=-1)
Accuracies [ 0.95681063  0.96494157  0.93791946]
```

The mean accuracy is comparable to the accuracy of the support vector classifier. Adding more hidden units or hidden layers and grid searching to tune the hyperparameters could further improve the accuracy.

# Summary

In this chapter, we introduced artificial neural networks, powerful models for classification and regression that can represent complex functions by composing several artificial neurons. In particular, we discussed directed acyclic graphs of artificial neurons called feedforward neural networks. Multilayer perceptrons are a type of feedforward network in which each layer is fully connected to the subsequent layer. An MLP with one hidden layer and a finite number of hidden units is a universal function approximator. It can represent any continuous function, though it will not necessarily be able to learn appropriate weights automatically. We described how the hidden layers of a network represent latent variables and how their weights can be learned using the backpropagation algorithm. Finally, we used scikit-learn's multilayer perceptron implementation to approximate the function XOR and to classify handwritten digits.

This chapter concludes the book. We discussed a variety of models, learning algorithms, and performance measures, as well as their implementations in scikit-learn. In the first chapter, we described machine learning programs as those that learn from experience to improve their performance at a task. Then, we worked through examples that demonstrated some of the most common experiences, tasks, and performance measures in machine learning. We regressed the prices of pizzas onto their diameters and classified spam and ham text messages. We clustered colors to compress images and clustered the SURF descriptors to recognize photographs of cats and dogs. We used principal component analysis for facial recognition, built a random forest to block banner advertisements, and used support vector machines and artificial neural networks for optical character recognition. Thank you for reading; I hope that you will be able to use scikit-learn and this book's examples to apply machine learning to your own experiences.

# Index

## D



## E



## F



## G



## H

## N



## O



## P



## Q



## R

## S



## T



## U



## V



## W



## X



## Z


## Thank you for buying
# Mastering Machine Learning with scikit-learn

## 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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

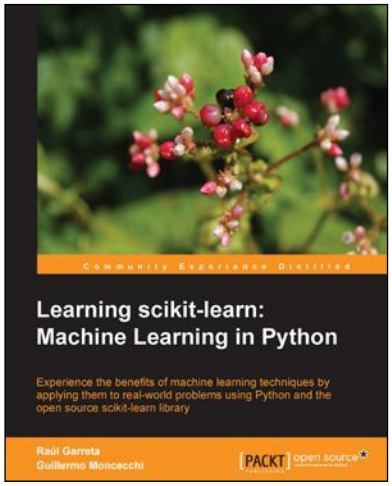
## About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, 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.



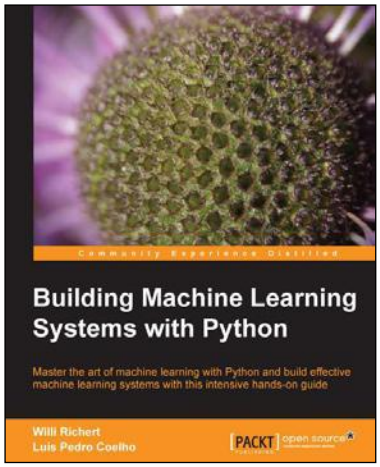

## Learning scikit-learn: Machine Learning in Python

ISBN: 978-1-78328-193-0 Paperback: 118 pages

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

1. Use Python and scikit-learn to create intelligent applications.
2. Apply regression techniques to predict future behavior and learn to cluster items in groups by their similarities.
3. Make use of classification techniques to perform image recognition and document classification.

---



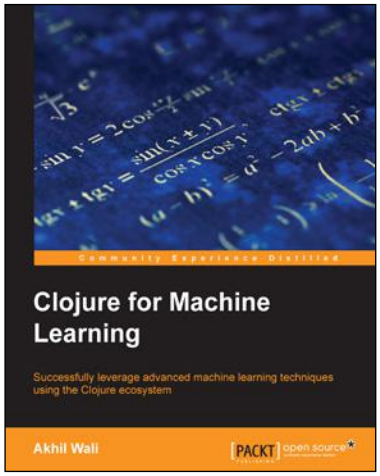

## Building Machine Learning Systems with Python

ISBN: 978-1-78216-140-0 Paperback: 290 pages

Master the art of machine learning with Python and build effective machine learning systems with this intensive hands-on guide

1. Master Machine Learning using a broad set of Python libraries and start building your own Python-based ML systems.
2. Covers classification, regression, feature engineering, and much more guided by practical examples.
3. A scenario-based tutorial to get into the right mind-set of a machine learner (data exploration) and successfully implement this in your new or existing projects.

[PACKT] open source
community experience distilled
PUBLISHING



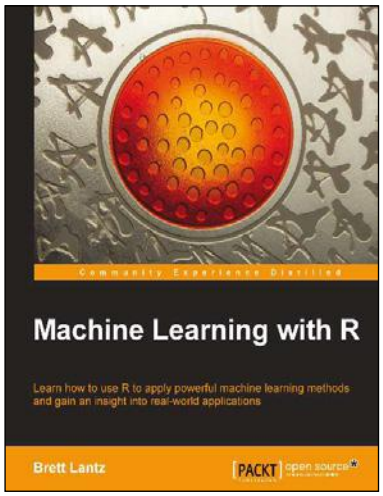

## Clojure for Machine Learning

ISBN: 978-1-78328-435-1 Paperback: 292 pages

Successfully leverage advanced machine learning techniques using the Clojure ecosystem

1. Covers a lot of machine learning techniques with Clojure programming.
2. Encompasses precise patterns in data to predict future outcomes using various machine learning techniques.
3. Packed with several machine learning libraries available in the Clojure ecosystem.

---




## Machine Learning with R

ISBN: 978-1-78216-214-8 Paperback: 396 pages

Learn how to use R to apply powerful machine learning methods and gain an insight into real-world applications

1. Harness the power of R for statistical computing and data science.
2. Use R to apply common machine learning algorithms with real-world applications.
3. Prepare, examine, and visualize data for analysis.
4. Understand how to choose between machine learning models.