If you have already studied machine learning theory in detail, this book will show you how to put your knowledge into practice. If you have used machine learning techniques before and want to gain more insight into how machine learning really works, this book is for you! Don't worry if you are completely new to the machine learning field; you have even more reason to be excited. I promise you that machine learning will change the way you think about the problems you want to solve and will show you how to tackle them by unlocking the power of data.

Before we dive deeper into the machine learning field, let me answer your most important question, "why Python?" The answer is simple: it is powerful yet very accessible. Python has become the most popular programming language for data science because it allows us to forget about the tedious parts of programming and offers us an environment where we can quickly jot down our ideas and put concepts directly into action.

Reflecting on my personal journey, I can truly say that the study of machine learning made me a better scientist, thinker, and problem solver. In this book, I want to share this knowledge with you. Knowledge is gained by learning, the key is our enthusiasm, and the true mastery of skills can only be achieved by practice. The road ahead may be bumpy on occasions, and some topics may be more challenging than others, but I hope that you will embrace this opportunity and focus on the reward. Remember that we are on this journey together, and throughout this book, we will add many powerful techniques to your arsenal that will help us solve even the toughest problems the data-driven way.

What this book covers

Chapter 1, *Giving Computers the Ability to Learn from Data*, introduces you to the main subareas of machine learning to tackle various problem tasks. In addition, it discusses the essential steps for creating a typical machine learning model building pipeline that will guide us through the following chapters.

Chapter 2, *Training Machine Learning Algorithms for Classification*, goes back to the origin of machine learning and introduces binary perceptron classifiers and adaptive linear neurons. This chapter is a gentle introduction to the fundamentals of pattern classification and focuses on the interplay of optimization algorithms and machine learning.

Classifiers

Chapter 3, *A Tour of Machine Learning Classifirs Using Scikit-learn*, describes the essential machine learning algorithms for classification and provides practical examples using one of the most popular and comprehensive open source machine learning libraries, scikit-learn.

Chapter 12, *Training Artificial Neural Networks for Image Recognition*, extends the concept of gradient-based optimization, which we first introduced in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, to build powerful, multilayer neural networks based on the popular backpropagation algorithm.

Chapter 13, *Parallelizing Neural Network Training with Theano*, builds upon the knowledge from the previous chapter to provide you with a practical guide for training neural networks more efficiently. The focus of this chapter is on Theano, an open source Python library that allows us to utilize multiple cores of modern GPUs.

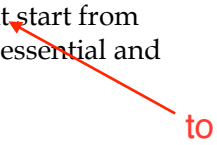
What you need for this book

The execution of the code examples provided in this book requires an installation of Python 3.4.3 or newer on Mac OS X, Linux, or Microsoft Windows. We will make frequent use of Python's essential libraries for scientific computing throughout this book, including SciPy, NumPy, scikit-learn, matplotlib, and pandas.

The first chapter will provide you with instructions and useful tips to set up your Python environment and these core libraries. We will add additional libraries to our repertoire and installation instructions are provided in the respective chapters: the NLTK library for natural language processing (*Chapter 8*, *Applying Machine Learning to Sentiment Analysis*), the Flask web framework (*Chapter 9*, *Embedding a Machine Learning Algorithm into a Web Application*), the seaborn library for statistical data visualization (*Chapter 10*, *Predicting Continuous Target Variables with Regression Analysis*), and Theano for efficient neural network training on graphical processing units (*Chapter 13*, *Parallelizing Neural Network Training with Theano*).

Who this book is for

If you want to find out how to use Python to start answering critical questions of your data, pick up *Python Machine Learning*—whether you want start from scratch or want to extend your data science knowledge, this is an essential and unmissable resource.



Conventions

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

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

More formally, we can pose this problem as a binary classification task where we refer to our two classes as `1` (positive class) and `-1` (negative class) for simplicity. We can then define an *activation function* $\phi(z)$ that takes a linear combination of certain input values $\boldsymbol{x}$ and a corresponding weight vector $\boldsymbol{w}$, where z is the so-called net input ($z = w_1x_1 + \ldots + w_mx_m$):

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

[...], otherwise. In [...]

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

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

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

In the following sections, we will often make use of basic notations from linear algebra. For example, we will abbreviate the sum of the products of the values in $\boldsymbol{x}$ and $\boldsymbol{w}$ using a *vector dot product*, whereas superscript **T** stands for *transpose*, which is an operation that transforms a column vector into a row vector and vice versa:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} \boldsymbol{x}_j \boldsymbol{w}_j = \boldsymbol{w}^T \boldsymbol{x}$$

For example: $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$.

Furthermore, the transpose operation can also be applied to a matrix to reflect it over its diagonal, for example:

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

In this book, we will only use the very basic concepts from linear algebra. However, if you need a quick refresher, please take a look at Zico Kolter's excellent Linear Algebra Review and Reference, which is freely available at `http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf`.

The following figure illustrates how the net input $z = \boldsymbol{w}^T \boldsymbol{x}$ is squashed into a binary output (-1 or 1) by the activation function of the perceptron (left subfigure) and how it can be used to discriminate between two linearly separable classes (right subfigure):

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

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

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

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

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

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

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

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

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

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

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

$$\hat{y}_j^{(i)} = +1,\ y^{(i)} = -1,\ \eta = 1$$

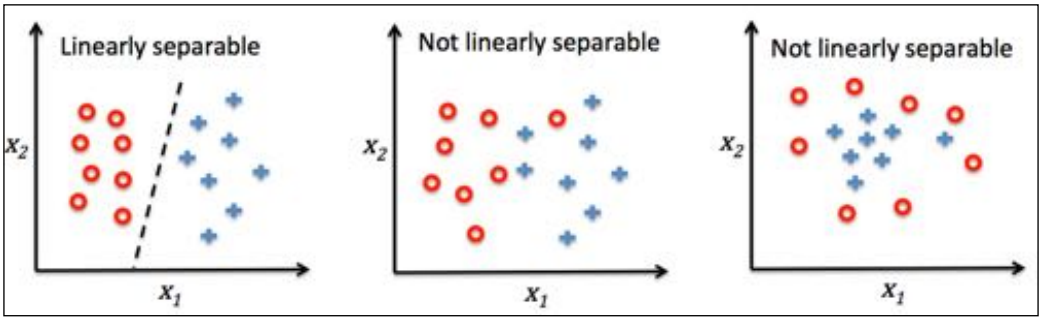
Let's assume that $x_j^{(i)} = 0.5$, and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1 so that the activation $x_j^{(i)} \times w_j^{(i)}$ will be more positive the next time we encounter this sample and thus will be more likely to be above the threshold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

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

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

$$\Delta w_j = \left(1^{(i)} - -1^{(i)}\right) 2^{(i)} = (2)\,2^{(i)} = 4$$

It is important to note that the convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small. If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (*epochs*) and/or a threshold for the number of tolerated misclassifications—the perceptron would never stop updating the weights otherwise:



Downloading the example code

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

If you are not yet familiar with Python's scientific libraries or need a refresher, please see the following resources:

NumPy: `http://wiki.scipy.org/Tentative_NumPy_Tutorial`

Pandas: `http://pandas.pydata.org/pandas-docs/stable/tutorials.html`

Matplotlib: `http://matplotlib.org/ussers/beginner.html`

Also, to better follow the code examples, I recommend you download the IPython notebooks from the Packt website. For a general introduction to IPython notebooks, please visit `https://ipython.org/ipython-doc/3/notebook/index.html`.

users

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

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

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

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

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

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

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

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

Training a perceptron model on the Iris dataset

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

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

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

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

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

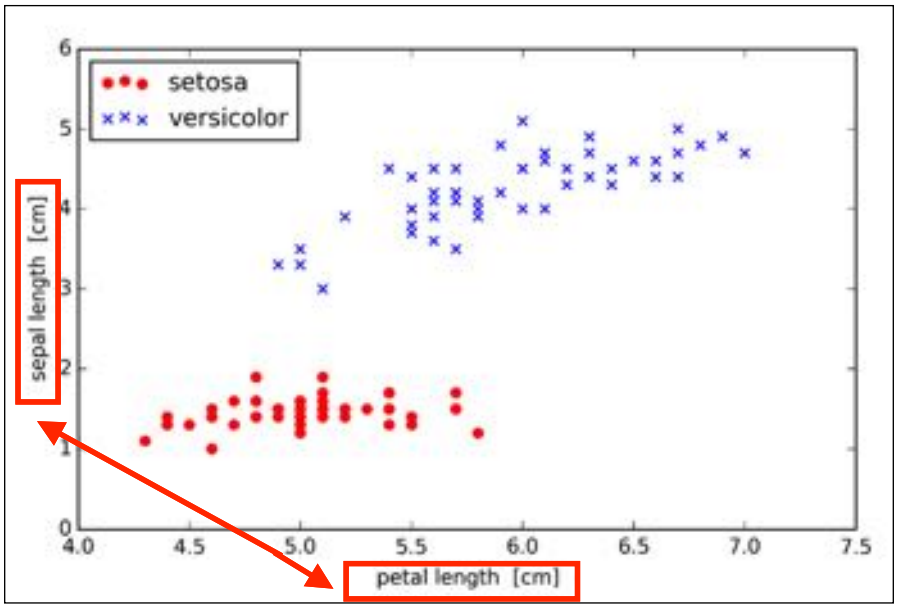
Next, we extract the first 100 class labels that correspond to the 50 *Iris-Setosa* and 50 *Iris-Versicolor* flowers, respectively, and convert the class labels into the two integer class labels `1` (*Versicolor*) and `-1` (*Setosa*) that we assign to a vector `y` where the values method of a pandas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

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

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

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

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



Now it's time to train our perceptron algorithm on the Iris data subset that we just extracted. Also, we will plot the `misclassification error` for each epoch to check if the algorithm converged and found a decision boundary that separates the two Iris flower classes:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_,
```

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

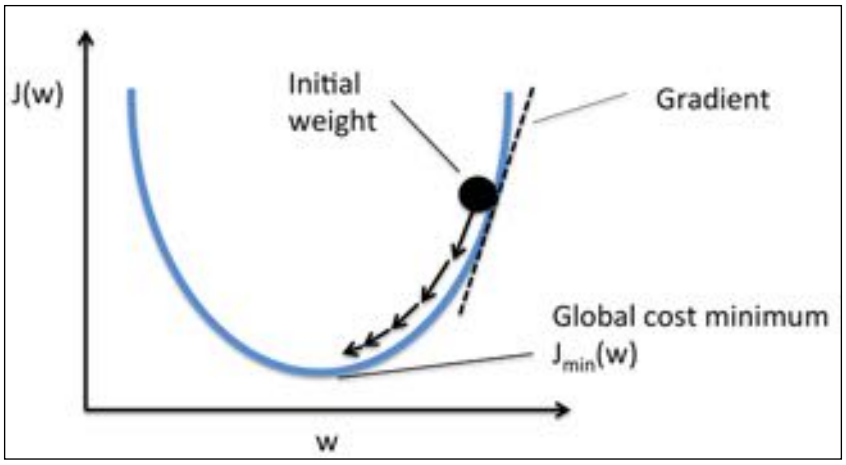
Minimizing cost functions with gradient descent

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

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

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

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



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

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$$

Here, the weight change $\Delta \boldsymbol{w}$ is defined as the negative gradient multiplied by the learning rate η:

$$\Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w}).$$

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

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

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

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

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

Now what we are actually interested in is predicting the probability that a certain sample belongs to a particular class, which is the inverse form of the logit function. It is also called the *logistic* function, sometimes simply abbreviated as *sigmoid* function due to its characteristic S-shape.

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

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

Now let's simply plot the sigmoid function for some values in the range -7 to 7 to see what it looks like:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.axhspan(0.0, 1.0, facecolor='1.0', alpha=1.0, ls='dotted')
>>> plt.axhline(y=0.5, ls='dotted', color='k')
>>> plt.yticks([0.0, 0.5, 1.0])
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi (z)$')
>>> plt.show()
```

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

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

$y^{(i)}$

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

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

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

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

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

To get a better grasp on this cost function, let's take a look at the cost that we calculate for one single-sample instance:

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

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

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

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

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

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

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

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

Here, λ is the so-called regularization parameter.

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

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

missing "y"

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

Via t
data
regul

ning
the

The
scikit
the t
whic

λ,

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

or

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

lower one prob. easier to read and more consistent w. page 60

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

So we can rewrite the regularized cost function of logistic regression as follows:

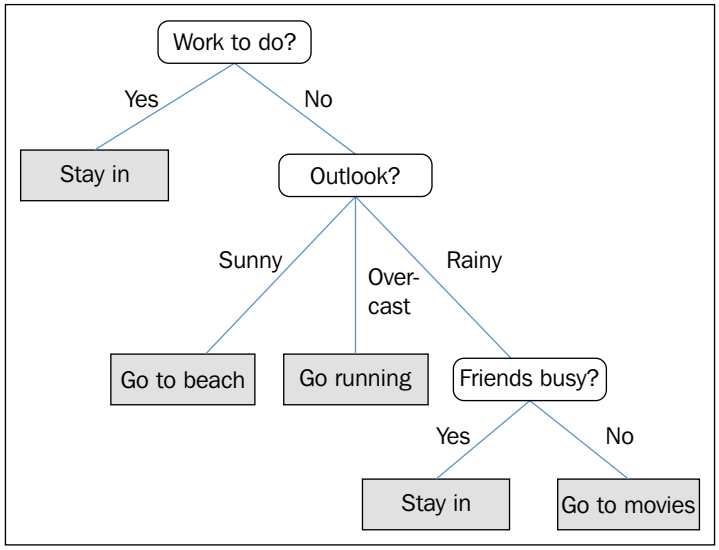
missing "y"

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

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

like above, also consider this alt. form for readability

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



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

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

Maximizing information gain – getting the most bang for the buck

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

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

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

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

Gini impurity

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

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

i

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

Gini impurity

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

(1 - p(i | t))

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

Gini impurity

Similar to entropy, the Gini index is maximal if the classes are perfectly mixed, for example, in a binary class setting ($c=2$):

$$1 - \sum_{i=1}^{c} 0.5^2 = 0.5$$

Gini impurity

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

Another impurity measure is the classification error:

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

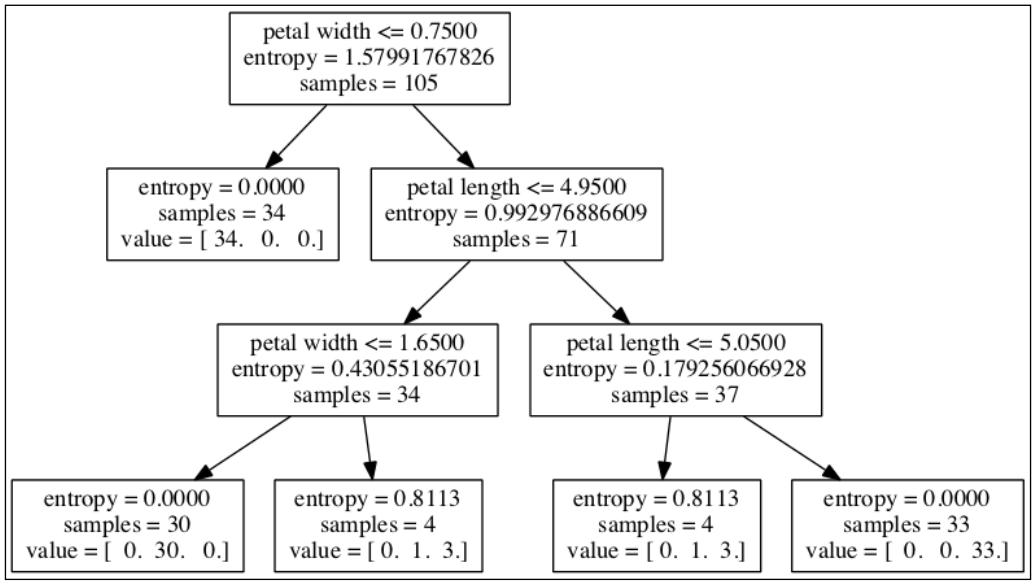
A nice feature in scikit-learn is that it allows us to export the decision tree as a `.dot` file after training, which we can visualize using the GraphViz program. This program is freely available at `http://www.graphviz.org` and supported by Linux, Windows, and Mac OS X.

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

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

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

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



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

petal width

Gini impurity

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

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

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

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

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

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

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

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

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

number swap

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

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

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

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

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

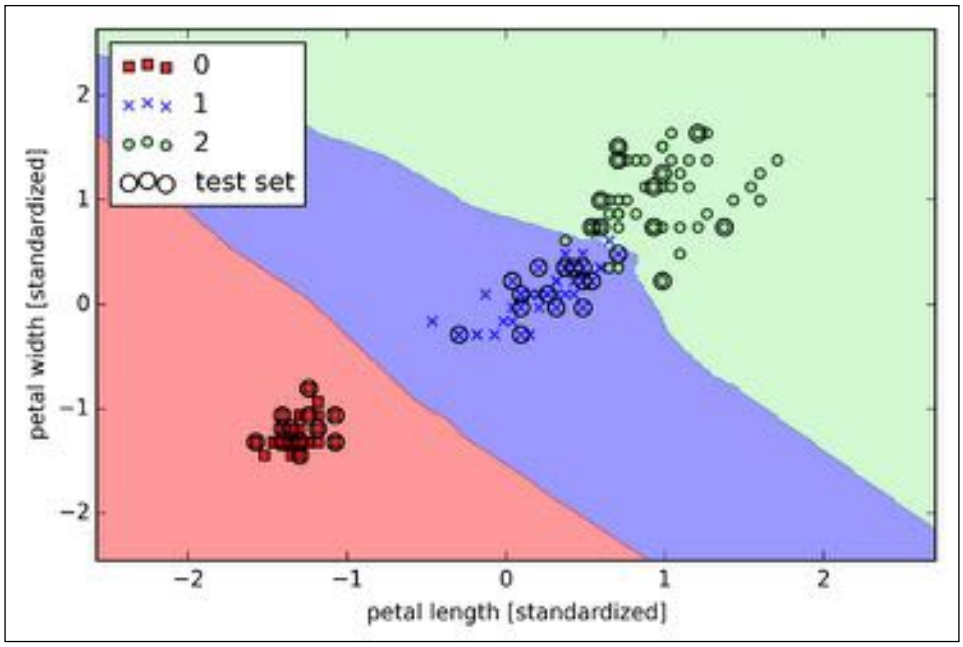
$$B: I_H\left(D_{right}\right) = 0$$

$$B: IG_H = 1 - \frac{6}{8}0.92 - 0 = 0.31$$

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

Gini impurity

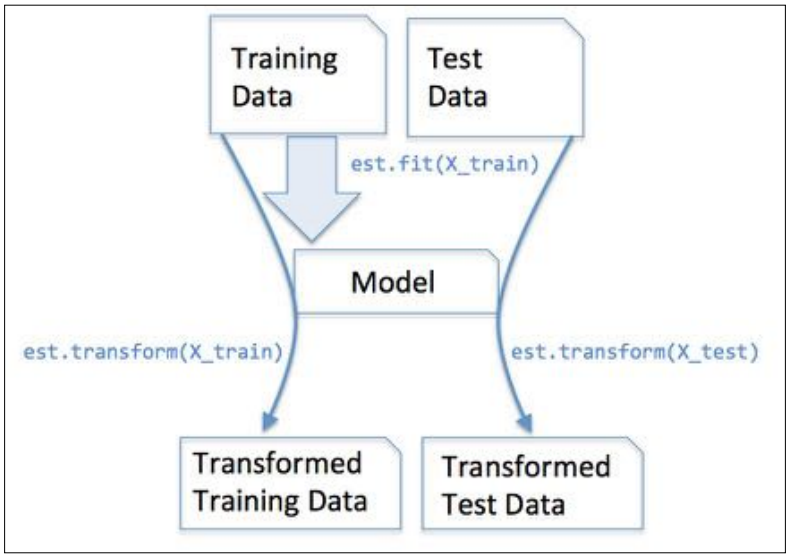
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>>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



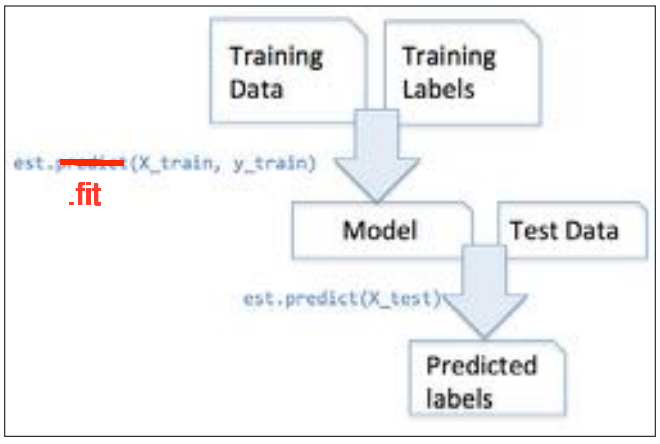
In the case of a tie, the scikit-learn implementation of the KNN algorithm will prefer the neighbors with a closer distance to the sample. If the neighbors have a similar distance, the algorithm will choose the class label that comes first in the training dataset.

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

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



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



Assessing feature importance with random forests

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

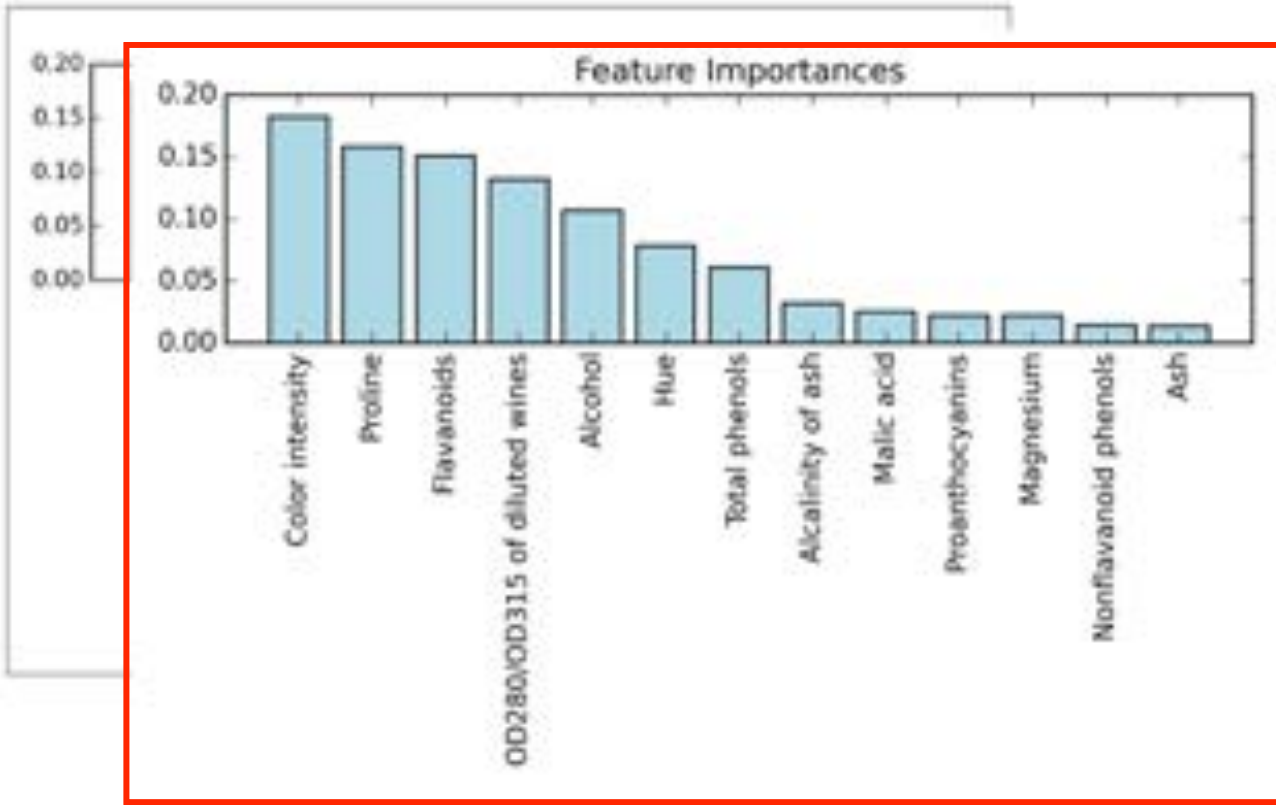
```
>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=10000,
...                                 random_state=0,
...                                 n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[f],
...                             importances[indices[f]]))
1) Alcohol                       0.182503
2) Malic acid                    0.158574
3) Ash                           0.150954
4) Alcalinity of ash             0.131983
5) Magnesium                     0.106564
6) Total phenols                 0.078249
7) Flavanoids                    0.060717
8) Nonflavanoid phenols          0.032039
9) Proanthocyanins               0.025385
10) Color intensity              0.022369
11) Hue                          0.022070
```

```
1) Color intensity               0.182483
2) Proline                       0.158610
3) Flavanoids                    0.150948
4) OD280/OD315 of diluted wines  0.131987
5) Alcohol                       0.106589
6) Hue                           0.078243
7) Total phenols                 0.060718
8) Alcalinity of ash             0.032033
9) Malic acid                    0.025400
10) Proanthocyanins              0.022351
11) Magnesium                    0.022078
12) Nonflavanoid phenols         0.014645
13) Ash                          0.013916
```

```
12) OD280/OD315 of diluted wines  0.014655
13) Proline                       0.013933
>>> plt.title('Feature Importances')
>>> plt.bar(range(X_train.shape[1]),
...         importances[indices],
...         color='lightblue',
...         align='center')
>>> plt.xticks(range(X_train.shape[1]),
...            feat_labels, rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()
```

feat_labels[indices]

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



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

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

Summary

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

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

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

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

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

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

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

Note that we want to re-use the training set parameters to transform any new data (or test data) as discussed in Chapter 3. I am sorry about this typo.
Please also see

https://github.com/rasbt/python-machine-learning-book/blob/master/faq/standardize-param-reuse.md

for an example why this can be a problem.

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

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

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

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

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

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

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

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

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

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

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

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

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

Insert this note

"Although the numpy.linalg.eig function was designed to decompose nonsymmetric square matrices, you may find that it returns complex eigenvalues in certain cases.
A related function, numpy.linalg.eigh, has been implemented to decompose Hermetian matrices, which is a numerically more stable approach to work

"In LDA, the number of linear discriminants is at most c-1 where c is the number of class labels, since the in-between class scatter matrix S is the sum of c matrices with rank 1 or less. We can indeed see ..."

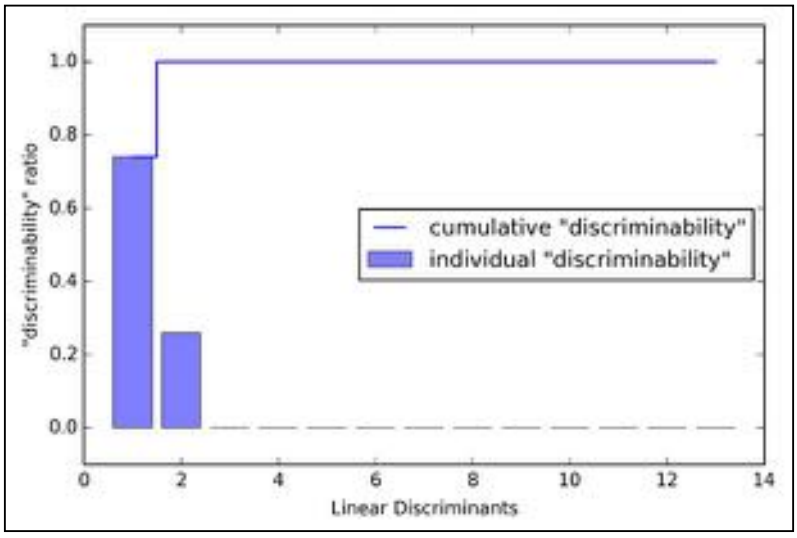
Replace text with

Those who are a little more familiar with linear algebra may know that the rank of the $d \times d$-dimensional covariance matrix can be at most $d-1$, and we can indeed see that we only have two nonzero eigenvalues (the eigenvalues 3-13 are not exactly zero, but this is due to the floating point arithmetic in NumPy). Note that in the rare case of perfect collinearity (all aligned sample points fall on a straight line), the covariance matrix would have rank one, which would result in only one eigenvector with a nonzero eigenvalue.

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

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

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



We do this for each pair of samples:

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

(n)

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

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

$$\boldsymbol{K}' = \boldsymbol{K} - 1_n \boldsymbol{K} - \boldsymbol{K} 1_n + 1_n \boldsymbol{K} 1_n$$

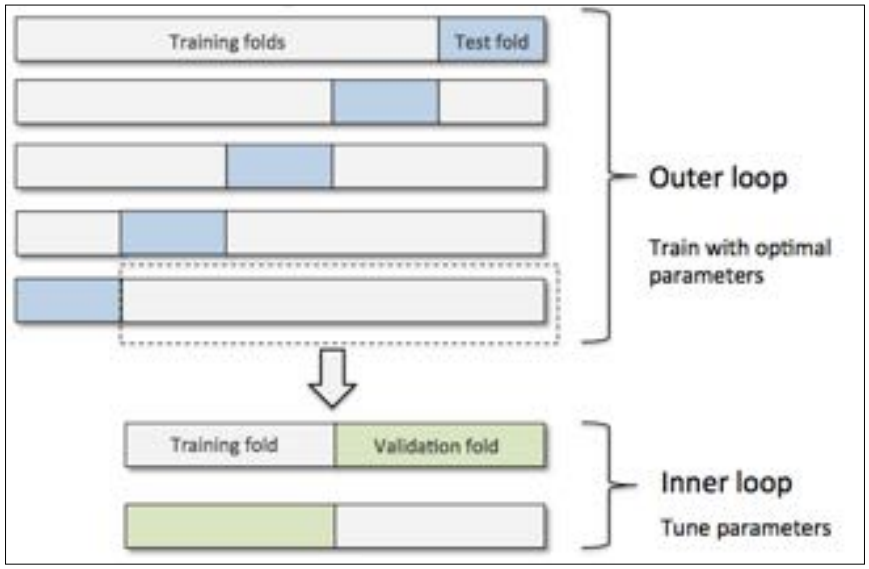
   Here, 1_n is an $n \times n$ - dimensional matrix (the same dimensions as the kernel matrix) where all values are equal to $\frac{1}{n}$.

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

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

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

In nested cross-validation, we have an outer k-fold cross-validation loop to split the data into training and test folds, and an inner loop is used to select the model using k-fold cross-validation on the training fold. After model selection, the test fold is then used to evaluate the model performance. The following figure explains the concept of nested cross-validation with five outer and two inner folds, which can be useful for large data sets where computational performance is important; this particular type of nested cross-validation is also known as **5x2 cross-validation**:



In scikit-learn, we can perform nested cross-validation as follows:

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                      cv=10, 5,
...                      n_jobs=-1) X_train, y_train
>>> scores = cross_val_score(gs, X, y, scoring='accuracy', cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...               np.mean(scores), np.std(scores)))
CV accuracy: 0.978 +/- 0.012
```

remove indent

Note: Optionally, you could use cv=2 here to produce the 5 x 2 nested CV that is shown in the figure.

note that the CV accuracy score is still correct (e.g., see code in the ipynb file)

Although the AdaBoost algorithm seems to be pretty straightforward, let's walk through a more concrete example using a training set consisting of 10 training samples as illustrated in the following table:

Sample indices	x	y	Weights	$\hat{y}(x <= 3.0)$?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

The first column of the table depicts the sample indices of the training samples 1 to 10. In the second column, we see the feature values of the individual samples assuming this is a one-dimensional dataset. The third column shows the true class label y_i for each training sample x_i, where $y \in \{1, -1\}$. The initial weights are shown in the fourth column; we initialize the weights to uniform and normalize them to sum to one. In the case of the 10 sample training set, we therefore assign the 0.1 to each weight w_i in the weight vector $\mathbf{w}$. The predicted class labels $\hat{y}$ are shown in the fifth column, assuming that our splitting criterion is $x \leq 3.0$. The last column of the table then shows the updated weights based on the update rules that we defined in the pseudocode.

Since the computation of the weight updates may look a little bit complicated at first, we will now follow the calculation step by step. We start by computing the weighted error rate ε as described in step 5:

$$\varepsilon = 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 = \frac{3}{10} = 0.3$$

$$\varepsilon = 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 1 + 0.1 \times 1 + 0.1 \times 1 + 0.1 \times 0 = 3/10 = 0.3$$

Next we compute the coefficient α_j (shown in step 6), which is later used in step 7 to update the weights as well as for the weights in majority vote prediction (step 10):

$$\alpha_j = \frac{0.5 \log(1-\varepsilon)}{\varepsilon} \approx 0.424 \quad \rightarrow \quad \alpha_j = 0.5 \log\left(\frac{1-\varepsilon}{\varepsilon}\right) \approx 0.424$$

Although we used another simple example for demonstration purposes, we can see that the performance of the AdaBoost classifier is slightly improved compared to the decision stump and achieved very similar accuracy scores to the bagging classifier that we trained in the previous section. However, we should note that it is considered ~~as~~ bad practice to select a model based on the repeated usage of the test set. The estimate of the generalization performance may be too optimistic, which we discussed in more detail in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Finally, let's check what the decision regions look like:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
...                         ['Decision Tree', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red',
...                        marker='o')
... axarr[idx].set_title(tt)
... axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s=Hue',
...          ha='center',
...          va='center',
...          fontsize=12)
>>> plt.show()
```

Now in order to calculate the tf-idf, we simply need to add 1 to the inverse document frequency and multiply it by the term frequency:

$$\text{tf-idf}\left("is", \text{d3}\right) = 2 \times \left(0 + 1\right) = 2$$

If we repeated these calculations for all terms in the 3rd document, we'd obtain the following tf-idf vectors: [1.69, 2.00, 1.29, 1.29, 1.29, 2.00, and 1.29]. However, we notice that the values in this feature vector are different from the values that we obtained from the `TfidfTransformer` that we used previously. The final step that we are missing in this tf-idf calculation is the L2-normalization, which can be applied as follows:

$$\text{tf-idf}\left("is", \text{d3}\right)_{norm} \frac{\left[1.69,\ 2.00,\ 1.29,\ 1.29,\ 1.29,\ 2.00,\ 1.29\right]}{\sqrt{1.69^2 + 2.00^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.00^2 + 1.29^2}}$$
$$= \left[0.40,\ 0.48,\ 0.31,\ 0.31,\ 0.31,\ 0.48,\ 0.31\right]$$

Here, it should be either
tf-idf("is", d3) = 0.48
or
tf-idf(d3) = [0.40, 0.48, 0.31, 0.31, 0.31, 0.48, 0.31]

As we can see, the results now match the results returned by scikit-learn's `TfidfTransformer`. Since we now understand how tf-idfs are calculated, let us proceed to the next sections and apply those concepts to the movie review dataset.

Cleaning text data

In the previous subsections, we learned about the bag-of-words model, term frequencies, and tf-idfs. However, the first important step—before we build our bag-of-words model—is to clean the text data by stripping it of all unwanted characters. To illustrate why this is important, let us display the last 50 characters from the first document in the reshuffled movie review dataset:

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

As we can see here, the text contains HTML markup as well as punctuation and other non-letter characters. While HTML markup does not contain much useful semantics, punctuation marks can represent useful, additional information in certain NLP contexts. However, for simplicity, we will now remove all punctuation marks but only keep **emoticon** characters such as ":)" since those are certainly useful for sentiment analysis. To accomplish this task, we will use Python's **regular expression** (**regex**) library, re, as shown here:

```
>>> import re
>>> def preprocessor(text):
```

The special case of one explanatory variable is also called **simple linear regression**, but of course we can also generalize the linear regression model to multiple explanatory variables. Hence, this process is called **multiple linear regression**:

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{i=0}^{n} w_i x_i = w^T x$$

Here, w_0 is the *y* axis intercept with $x_0 = 1$.

Exploring the Housing Dataset

Before we implement our first linear regression model, we will introduce a new dataset, the **Housing Dataset**, which contains information about houses in the suburbs of Boston collected by D. Harrison and D.L. Rubinfeld in 1978. The *Housing Dataset* has been made freely available and can be downloaded from the *UCI machine learning repository* at `https://archive.ics.uci.edu/ml/datasets/Housing`.

The features of the 506 samples may be summarized as shown in the excerpt of the dataset description:

- **CRIM**: This is the per capita crime rate by town
- **ZN**: This is the proportion of residential land zoned for lots larger than 25,000 sq.ft.
- **INDUS**: This is the proportion of non-retail business acres per town
- **CHAS**: This is the Charles River dummy variable (this is equal to 1 if tract bounds river; 0 otherwise)
- **NOX**: This is the nitric oxides concentration (parts per 10 million)
- **RM**: This is the average number of rooms per dwelling
- **AGE**: This is the proportion of owner-occupied units built prior to 1940
- **DIS**: This is the weighted distances to five Boston employment centers
- **RAD**: This is the index of accessibility to radial highways
- **TAX**: This is the full-value property-tax rate per $10,000
- **PTRATIO**: This is the pupil-teacher ratio by town
- **B**: This is calculated as *1000(Bk - 0.63)^2*, where Bk is the proportion of people of African American descent by town
- **LSTAT**: This is the percentage lower status of the population
- **MEDV**: This is the median value of owner-occupied homes in $1000s

On a side note, it is also worth mentioning that we technically don't have to update the weights of the intercept if we are working with standardized variables since the *y* axis intercept is always 0 in those cases. We can quickly confirm this by printing the weights:

```
>>> print('Slope: %.3f' % lr.w_[1])
Slope: 0.695
>>> print('Intercept: %.3f' % lr.w_[0])
Intercept: -0.000
```

Estimating the coefficient of a regression model via scikit-learn

In the previous section, we implemented a working model for regression analysis. However, in a real-world application, we may be interested in more efficient implementations, for example, scikit-learn's `LinearRegression` object that makes use of the **LIBLINEAR** library and advanced optimization algorithms that work better with unstandardized variables. This is sometimes desirable for certain applications:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> print('Slope: %.3f' % slr.coef_[0])
Slope: 9.102
>>> print('Intercept: %.3f' % slr.intercept_)
Intercept: -34.671
```

As we can see by executing the preceding code, scikit-learn's `LinearRegression` model fitted with the unstandardized **RM** and **MEDV** variables yielded different model coefficients. Let's compare it to our own GD implementation by plotting MEDV against RM:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Average number of rooms [RM] (standardized)')
>>> plt.ylabel('Price in $1000\'s [MEDV] (standardized)')
>>> plt.show()
```

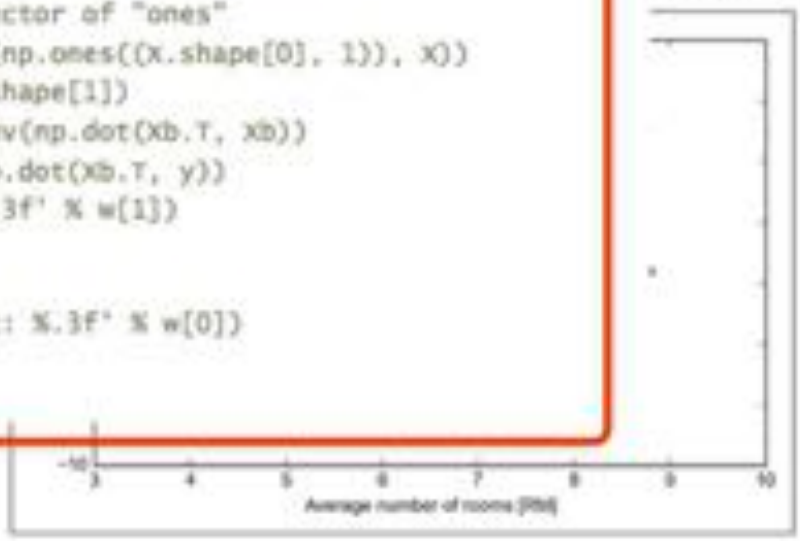
closed-form solution for solving OLS involving a system of linear equations that can be found in most introductory statistics textbooks:

$$\mathbf{w} = (X^T X)^{-1} X^T y$$

We can implement it in Python as follows:

```
# adding a column vector of "ones"
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print('Slope: %.3f' % w[1])
Slope: 9.102
>>> print('Intercept: %.3f' % w[0])
Intercept: -34.671
```

executing the code
3D implementation:



replace section with...

As an alternative to using machine learning libraries, there is a closed-form solution for solving OLS involving a system of linear equations that can be found in most introductory statistics textbooks:

$$w_1 = \left(X^T X\right)^{-1} X^T y$$

$$w_0 = \mu_y - \mu_x \mu_1$$

Here, μ_y is the mean of the true target values and μ_x is the mean of the predicted response.

The advantage of this method is that it is guaranteed to find the optimal solution analytically. However, if we are working with very large datasets, it can be computationally too expensive to invert the matrix in this formula (sometimes also called the **normal equation**) or the sample matrix may be singular (non-invertible), which is why we may prefer iterative methods in certain cases.

If you are interested in more information on how to obtain the normal equations, I recommend you take a look at Dr. Stephen Pollock's chapter, *The Classical Linear Regression Model* from his lectures at the University of Leicester, which are available for free at `http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf`.

However, a limitation of the LASSO is that it selects at most n variables if $m > n$. A compromise between Ridge regression and the LASSO is the Elastic Net, which has a L1 penalty to generate sparsity and a L2 penalty to overcome some of the limitations of the LASSO, such as the number of selected variables.

$$J(w)_{ElasticNet} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1 \sum_{j=1}^{m} w_j^2 + \lambda_2 \sum_{j=1}^{m} |w_j|$$

Those regularized regression models are all available via scikit-learn, and the usage is similar to the regular regression model except that we have to specify the regularization strength via the parameter λ, for example, optimized via k-fold cross-validation.

A Ridge Regression model can be initialized as follows:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Note that the regularization strength is regulated by the parameter `alpha`, which is similar to the parameter λ. Likewise, we can initialize a LASSO regressor from the `linear_model` submodule:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Lastly, the `ElasticNet` implementation allows us to vary the L1 to L2 ratio:

```
>>> from sklearn.linear_model import ElasticNet
>>> lasso = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

For example, if we set `l1_ratio` to `1.0`, the `ElasticNet` regressor would be equal to LASSO regression. For more detailed information about the different implementations of linear regression, please see the documentation at `http://scikit-learn.org/stable/modules/linear_model.html`.

Turning a linear regression model into a curve – polynomial regression

In the previous sections, we assumed a linear relationship between explanatory and response variables. One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms:

$$y = w_0 + w_1 x + w_2 x^2 x^2 + \ldots + w_d x^d$$

After executing the preceding code, we should now see the following ~~distance matrix~~:

DataFrame containing the randomly generated samples:

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

Performing hierarchical clustering on a distance matrix

To calculate the distance matrix as input for the hierarchical clustering algorithm, we will use the `pdist` function from SciPy's `spatial.distance` submodule:

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...             pdist(df, metric='euclidean')),
...             columns=labels, index=labels)
>>> row_dist
```

Using the preceding code, we calculated the Euclidean distance between each pair of sample points in our dataset based on the features X, Y, and Z. We provided the condensed distance matrix—returned by `pdist`—as input to the `squareform` function to create a symmetrical matrix of the pair-wise distances, as shown here:

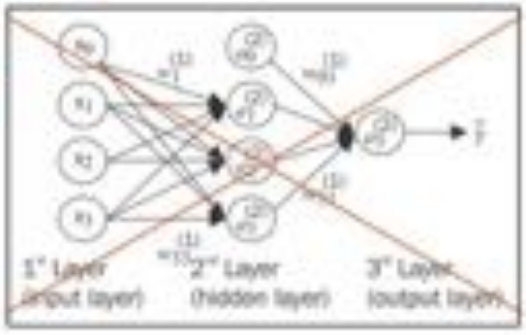
	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

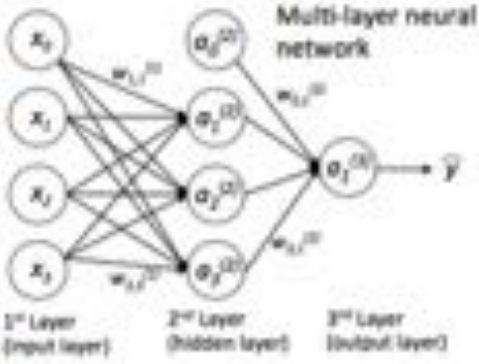
Note that although Adaline consists of two layers, one input layer and one output layer, it is called a single-layer network because of its single link between the input and output layers.

Introducing the multi-layer neural network architecture

In this section, we will see how to connect multiple single neurons to a **multi-layer feedforward neural network**; this special type of network is also called a **multi-layer perceptron** (**MLP**). The following figure explains the concept of an MLP consisting of three layers: one input layer, one **hidden layer**, and one output layer. The units in the hidden layer are fully connected to the input layer, and the output layer is fully connected to the hidden layer, respectively. If such a network h
hidden layer, we also call it a *deep* artificial neural network.

Unfortunately, a lot of typos were made in this figure, please refer to my original to the right



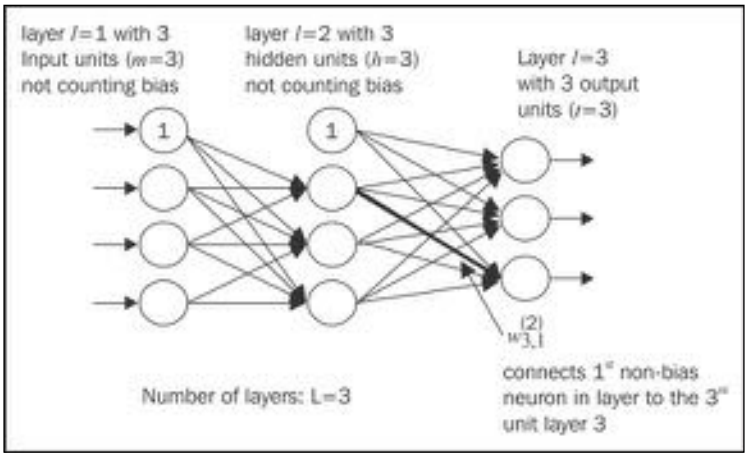


We could add an arbitrary number of hidden layers to the MLP to create deeper network architectures. Practically, we can think of the number of layers and units in a neural network as additional **hyperparameters** that we want to optimize for a given problem task using the cross-validation that we discussed in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

However, the error gradients that we will calculate later via backpropagation would become increasingly small as more layers are added to a network. This *vanishing gradient* problem makes the model learning more challenging. Therefore, special algorithms have been developed to pretrain such deep neural network structures, which is called *deep learning*.

If you are new to neural network representations, the terminology around the indices (subscripts and superscripts) may look a little bit confusing at first. You may wonder why we wrote $w_{j,k}^{(l)}$ and not $w_{k,j}^{(l)}$ to refer to the weight coefficient that connects the k^{th} unit in layer l to the j^{th} unit in layer $l+1$. What may seem a little bit quirky at first will make much more sense in later sections when we vectorize the neural network representation. For example, we will summarize the weights that connect the input and hidden layer by a matrix $\boldsymbol{W}^{(1)} \in \mathbb{R}^{h \times [m+1]}$, where h is the number of hidden units and $m+1$ is the number of hidden units plus bias unit. Since it is important to internalize this notation to follow the concepts later in this chapter, let's summarize what we just discussed in a descriptive illustration of a simplified 3-4-3 multi-layer perceptron:

input units



Activating a neural network via forward propagation

In this section, we will describe the process of **forward propagation** to calculate the output of an MLP model. To understand how it fits into the context of learning an MLP model, let's summarize the MLP learning procedure in three simple steps:

1. Starting at the input layer, we forward propagate the patterns of the training data through the network to generate an output.
2. Based on the network's output, we calculate the error that we want to minimize using a cost function that we will describe later.
3. We backpropagate the error, find its derivative with respect to each weight in the network, and update the model.

Finally, after repeating the steps for multiple epochs and learning the weights of the MLP, we use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels in the one-hot representation, which we described in the previous section.

Now, let's walk through the individual steps of forward propagation to generate an output from the patterns in the training data. Since each unit in the hidden unit is connected to all units in the input layers, we first calculate the activation $a_1^{(2)}$ as follows:

$$z_1^{(2)} = a_0^{(1)} w_{1,0}^{(1)} + a_1^{(1)} w_{1,1}^{(1)} + \cdots + a_m^{(1)} w_{1,m}^{(1)}$$

$$a_1^{(2)} = \phi\left(z_1^{(2)}\right)$$

Here, $z_1^{(2)}$ is the net input and $\phi(\cdot)$ is the activation function, which has to be differentiable to learn the weights that connect the neurons using a gradient-based approach. To be able to solve complex problems such as image classification, we need nonlinear activation functions in our MLP model, for example, the **sigmoid** (**logistic**) activation function that we used in **logistic regression** in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

As we can remember, the sigmoid function is an S-shaped curve that maps the net input z onto a logistic distribution in the range 0 to 1, ~~which passes the origin at $z = 0.5$~~, as shown in the following graph:

[...], which cuts the y-axis at z=0, [...]

Everywhere you read "h" on this page, you can think of "h" as "h+1" to include the bias unit (and in order to get the dimensions right)

Here, $\boldsymbol{a}^{(1)}$ is our $[m+1]\times 1$ dimensional feature vector of a sample $\boldsymbol{x}^{(i)}$ plus bias unit. $\boldsymbol{W}^{(1)}$ is an $h\times[m+1]$ dimensional weight matrix where h is the number of hidden units in our neural network. After matrix-vector multiplication, we obtain the $h\times 1$ dimensional net input vector $\boldsymbol{z}^{(2)}$ to calculate the activation $\boldsymbol{a}^{(2)}$ (where $\boldsymbol{a}^{(2)} \in \mathbb{R}^{h\times 1}$). Furthermore, we can generalize this computation to all n samples in the training set:

$$\boldsymbol{Z}^{(2)} = \boldsymbol{W}^{(1)}\left[\boldsymbol{A}^{(1)}\right]^T$$

Here, $\boldsymbol{A}^{(1)}$ is now an $n\times[m+1]$ matrix, and the matrix-matrix multiplication will result in a $h\times n$ dimensional net input matrix $\boldsymbol{Z}^{(2)}$. Finally, we apply the activation function $\phi(\cdot)$ to each value in the net input matrix to get the $h\times n$ activation matrix $\boldsymbol{A}^{(2)}$ for the next layer (here, output layer):

$$\boldsymbol{A}^{(2)} = \phi\left(\boldsymbol{Z}^{(2)}\right)$$

Similarly, we can rewrite the activation of the output layer in the vectorized form:

$$\boldsymbol{Z}^{(3)} = \boldsymbol{W}^{(2)}\boldsymbol{A}^{(2)}$$

Here, we multiply the $t\times h$ matrix $\boldsymbol{W}^{(2)}$ (t is the number of output units) by the $h\times n$ dimensional matrix $\boldsymbol{A}^{(2)}$ to obtain the $t\times n$ dimensional matrix $\boldsymbol{Z}^{(3)}$ (the columns in this matrix represent the outputs for each sample).

Lastly, we apply the sigmoid activation function to obtain the continuous valued output of our network:

$$\boldsymbol{A}^{(3)} = \phi\left(\boldsymbol{Z}^{(3)}\right), \quad \boldsymbol{A}^{(3)} \in \mathbb{R}^{t\times n}$$

Classifying handwritten digits

In the previous section, we covered a lot of the theory around neural networks, which can be a little bit overwhelming if you are new to this topic. Before we continue with the discussion of the algorithm for learning the weights of the MLP model, backpropagation, let's take a short break from the theory and see a neural network in action.

```
            grad2[:, 1:] += (w2[:, 1:] * (self.l1 + self.l2))

            return grad1, grad2

        def predict(self, X):
            a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)
            y_pred = np.argmax(z3, axis=0)
            return y_pred

        def fit(self, X, y, print_progress=False):
            self.cost_ = []
            X_data, y_data = X.copy(), y.copy()
            y_enc = self._encode_labels(y, self.n_output)

            delta_w1_prev = np.zeros(self.w1.shape)
            delta_w2_prev = np.zeros(self.w2.shape)

            for i in range(self.epochs):

                # adaptive learning rate
                self.eta /= (1 + self.decrease_const*i)

                if print_progress:
                    sys.stderr.write(
                            '\rEpoch: %d/%d' % (i+1, self.epochs))
                    sys.stderr.flush()

                if self.shuffle:
                    idx = np.random.permutation(y_data.shape[0])
                    X_data, y_data = X_data[idx], y_data[idx]

                mini = np.array_split(range(
                             y_data.shape[0]), self.minibatches)
                for idx in mini:

                    # feedforward
                    a1, z2, a2, z3, a3 = self._feedforward(
                                         X[idx], self.w1, self.w2)
                    cost = self._get_cost(y_enc=y_enc[:, idx],
                                          output=a3,
                                          w1=self.w1,
                                          w2=self.w2)
                    self.cost_.append(cost)
```

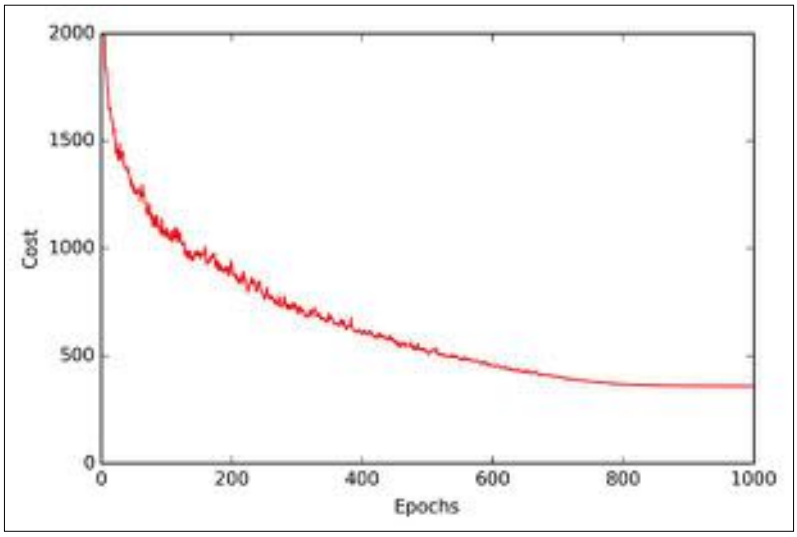
X_data, y_enc = X_data[idx], y_enc[:,idx

X_data[idx]


```
>>> nn = NeuralNetMLP([...],
...                   [...],
...                   shuffle=False,
...                   random_state=1)
```

These line changes above enable shuffling if the setting is `shuffle=True`.
To match the original output in the book (no shuffling) after applying this patch, the `shuffle=False` setting needs to be added when the NeuralNetMLP is initialized (next page) as shown on the left

The following plot gives us a clearer picture indicating that the training algorithm converged shortly after the 800th epoch:



Now, let's evaluate the performance of the model by calculating the prediction accuracy:

```
>>> y_train_pred = nn.predict(X_train)
>>> acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (acc * 100))
Training accuracy: 97.74%
```

As we can see, the model classifies most of the training digits correctly, but how does it generalize to data that it has not seen before? Let's calculate the accuracy on 10,000 images in the test dataset:

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
>>> print('Training accuracy: %.2f%%' % (acc * 100))
Test accuracy: 96.18%
```

Test

Based on the small discrepancy between training and test accuracy, we can conclude that the model only slightly overfits the training data. To further fine-tune the model, we could change the number of hidden units, values of the regularization parameters, learning rate, values of the decrease constant, or the adaptive learning using the techniques that we discussed in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning* (this is left as an exercise for the reader).

Although our MLP implementation supports both L1 and L2 regularization, we will now only focus on the L2 regularization term for simplicity. However, the same concepts apply to the L1 regularization term. By adding the L2 regularization term to our logistic cost function, we obtain the following equation:

$$J(\boldsymbol{w}) = \left[\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)\right] + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$$

Since we implemented an MLP for multi-class classification, this returns an output vector of t elements, which we need to compare with the $t \times 1$ dimensional target vector in the one-hot encoding representation. For example, the activation of the third layer and the target class (here: class 2) for a particular sample may look like this:

$$a^{(3)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Thus, we need to generalize the logistic cost function to all activation units j in our network. So our cost function (without the regularization term) becomes:

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n}\sum_{k=1}^{t} y_j^{(i)} \log\left(a_j^{(i)}\right) + \left(1 - y_j^{(i)}\right) \log\left(1 - a_j^{(i)}\right)$$

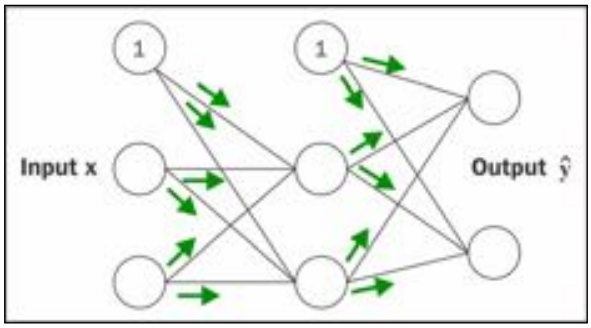
j=1

Here, the superscript i is the index of a particular sample in our training set.

The following generalized regularization term may look a little bit complicated at first, but here we are just calculating the sum of all weights of a layer l (without the bias term) that we added to the first column:

$$J(\boldsymbol{w}) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{(i)} \log\left(\phi\left(z_j^{(i)}\right)\right) + \left(1 - y_j^{(i)}\right) \log\left(1 - \phi\left(z_j^{(i)}\right)\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{ul}\sum_{j=1}^{ul+1}\left(w_{j,i}^{(l)}\right)^2$$

Concisely, we just forward propagate the input features through the connection in the network as shown here:



In backpropagation, we propagate the error from right to left. We start by calculating the error vector of the output layer:

$$\delta^{(3)} = a^{(3)} - y$$

Here, y is the vector of the true class labels.

Next, we calculate the error term of the hidden layer:

$$\delta^{(2)} = \left(W^{(2)}\right)^{T} \delta^{(3)} * \frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}}$$

Here, $\frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}}$ is simply the derivative of the sigmoid activation function, which we implemented as `_sigmoid_gradient`:

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

Note that the asterisk symbol $(*)$ means element-wise multiplication in this context.

Alternatively, you can apply these settings only to a particular Python script, by running it as follows:

```
THEANO_FLAGS=floatX=float32 python your_script.py
```

So far, we discussed how to set the default floating-point types to get the best bang for the buck on our GPU using Theano. Next, let's discuss the options to toggle between CPU and GPU execution. If we execute the following code, we can check whether we are using CPU or GPU:

```
>>> print(theano.config.device)
cpu
```

My personal recommendation is to use `cpu` as default, which makes prototyping and code debugging easier. For example, you can run Theano code on your CPU by executing it a script, as from your command-line terminal:

```
THEANO_FLAGS=device=cpu,floatX=float64 python your_script.py
```

However, once we have implemented the code and want to run it most efficiently utilizing our GPU hardware, we can then run it via the following code without making additional modifications to our original code:

```
THEANO_FLAGS=device=gpu,floatX=float32 python your_script.py
```

It may also be convenient to create a `.theanorc` file in your home directory to make these configurations permanent. For example, to always use `float32` and the GPU, you can create such a `.theanorc` file including these settings. The command is as follows:

```
echo -e "\n[global]\nfloatX=float32\ndevice=gpu\n" >> ~/.theanorc
```

If you are not operating on a MacOS X or Linux terminal, you can create a `.theanorc` file manually using your favorite text editor and add the following contents:

```
[global]
floatX=float32
device=gpu
```

Now that we know how to configure Theano appropriately with respect to our available hardware, we can discuss how to use more complex array structures in the next section.

Choosing activation functions for feedforward neural networks

For simplicity, we have only discussed the sigmoid activation function in context of multilayer feedforward neural networks so far; we used it in the hidden layer as well as the output layer in the multilayer perceptron implementation in *Chapter 12*, *Training Artificial Neural Networks for Image Recognition*. Although we referred to this activation function as *sigmoid* function—as it is commonly called in literature—the more precise definition would be *logistic function* or *negative log-likelihood function*. In the following subsections, you will learn more about alternative sigmoidal functions that are useful for implementing multilayer neural networks.

Technically, we could use any function as activation function in multilayer neural networks as long as it is differentiable. We could even use linear activation functions such as in Adaline (*Chapter 2*, *Training Machine Learning Algorithms for Classification*). However, in practice, it would not be very useful to use linear activation functions for both hidden and output layers, since we want to introduce nonlinearity in a typical artificial neural network to be able to tackle complex problem tasks. The sum of linear functions yields a linear function after all.

The logistic activation function that we used in the previous chapter probably mimics the concept of a neuron in a brain most closely: we can think of it as probability of whether a neuron fires or not. However, logistic activation functions can be problematic if we have highly negative inputs, since the output of the sigmoid function would be close to zero in this case. If the sigmoid function returns outputs that are close to zero, the neural network would learn very slowly and it becomes more likely that it gets trapped in local minima during training. This is why people often prefer a **hyperbolic tangent** as activation function in hidden layers. Before we discuss what a hyperbolic tangent looks like, let's briefly recapitulate some of the basics of the logistic function and look at a generalization that makes it more useful for multi-class classification tasks.

As we can see, the predicted class probabilities now sum up to one, as we would expect. It is also notable that the probability for the second class is close to zero, since there is a large gap between z_1 and $\max(z)$. However, note that the predicted class label is the same as in the logistic function. Intuitively, it may help to think of the softmax function as a *normalized* logistic function that is useful to obtain meaningful class-membership predictions in multi-class settings.

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('predicted class label:
...        %d' % y_class[0])
predicted class label: 2
```

Broadening the output spectrum by using a hyperbolic tangent

Another sigmoid function that is often used in the hidden layers of artificial neural networks is the **hyperbolic tangent** (**tanh**), which can be interpreted as a rescaled version of the logistic function.

$$\phi_{\text{tanh}}(z) = 2 \times \phi_{logistic}(2 \times z) - 1 = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$\phi_{logistic}(z) = \frac{1}{1 + e^{-z}}$$

~~$\text{logistic}(2 \times z) \times 2 - 1$~~

However, in certain contexts, it can be useful to return meaningful class probabilities for multi-class predictions. In the next section, we will take a look at a generalization of the logistic function, the **softmax** function, which can help us with this task.

Estimating probabilities in multi-class classification via the softmax function

The **softmax** function is a generalization of the logistic function that allows us to compute meaningful class-probabilities in multi-class settings (multinomial logistic regression). In softmax, the probability of a particular sample with net input z belongs to the i th class can be computed with a normalization term in the denominator that is the sum of all M linear functions:

$$P(y = i \mid z) = \phi_{softmax}(z) = \frac{e^{z}_{i}}{\sum_{m=1}^{M} e^{z}_{m}}$$

To see softmax in action, let's code it up in Python:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))

>>> def softmax_activation(X, w):
...     z = net_input(X, w)
...     return ~~sigmoid(z)~~ softmax(z)

>>> y_probas = softmax(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
 [[ 0.40386493]
 [ 0.07756222]
 [ 0.51857284]]
>>> y_probas.sum()
1.0
```

In the last two chapters of this book, we caught a glimpse of the most beautiful and most exciting algorithms in the whole machine learning field: artificial neural networks. Although deep learning really is beyond the scope of this book, I hope I could at least kindle your interest to follow the most recent advancement in this field. If you are considering a career as a machine learning researcher, or even if you just want to keep up to date with the current advancement in this field, I can recommend you to follow the works of the leading experts in this field, such as Geoff Hinton (`http://www.cs.toronto.edu/~hinton/`), Andrew Ng (`http://www.andrewng.org`), Yann LeCun (`http://yann.lecun.com`), Juergen Schmidhuber (`http://people.idsia.ch/~juergen/`), and Yoshua Bengio (`http://www.iro.umontreal.ca/~bengioy`), just to name a few. Also, please do not hesitate to join the scikit-learn, Theano, and Keras mailing lists to participate in interesting discussions around these libraries, and machine learning in general. I am looking forward to ~~meet~~ meeting you there! You are always welcome to contact me if you have any questions about this book or need some general tips about machine learning.

I hope this journey through the different aspects of machine learning was really worthwhile, and you learned many new and useful skills to advance your career and apply them to real-world problem solving.