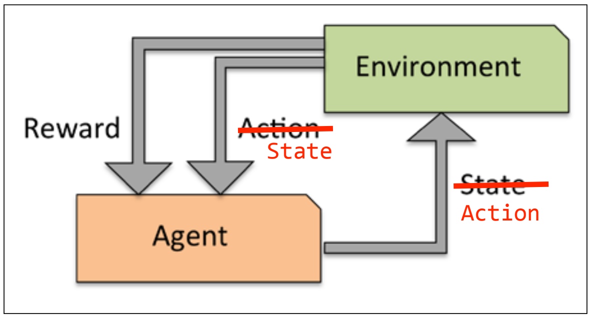
Solving interactive problems with reinforcement learning

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

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



Discovering hidden structures with unsupervised learning

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

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

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

remove parenthesis around “m” subscript

For example, x_1^{150} refers to the first dimension of flower sample 150, the *sepal* ~~*width.*~~ length Thus, each row in this feature matrix represents one flower instance and can be written as four-dimensional column vector $\boldsymbol{x}^{(i)} \in \mathbb{R}^{\mathbf{1\times 4}}$, $\boldsymbol{x}^{(i)} = \left[x_1^{(i)} \quad x_2^{(i)} \quad x_3^{(i)} \quad x_4^{(i)}\right]$.

Each feature dimension is a 150-dimensional row vector $\cancel{\boldsymbol{x}^{(i)}}$ x_j $\in \mathbb{R}^{\mathbf{150\times 1}}$, for example:

$$\boldsymbol{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}.$$

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

A roadmap for building machine learning systems

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

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

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

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

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

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

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

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

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

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

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

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

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

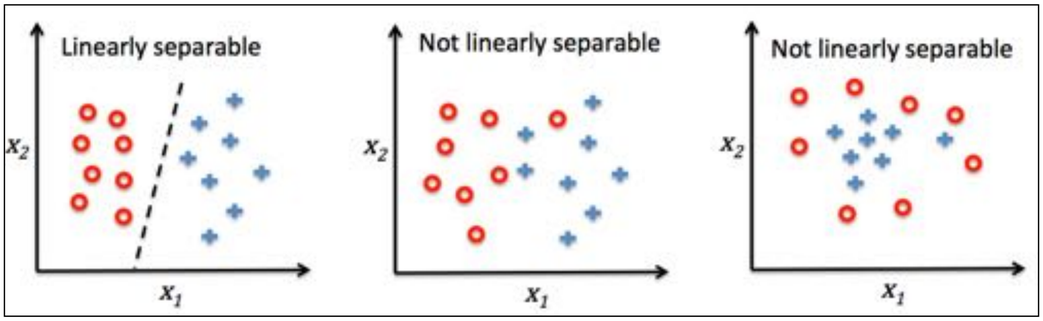
Let's assume that $x_j^{(i)} = 0.5$, and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1 so that the ~~activation~~ net input $x_j^{(i)} \overset{\times}{\cancel{=}} w_j^{(i)}$ will be more positive the next time we encounter this sample and thus will be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

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

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:

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

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.

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 $\boldsymbol{w}^T \boldsymbol{x}$.

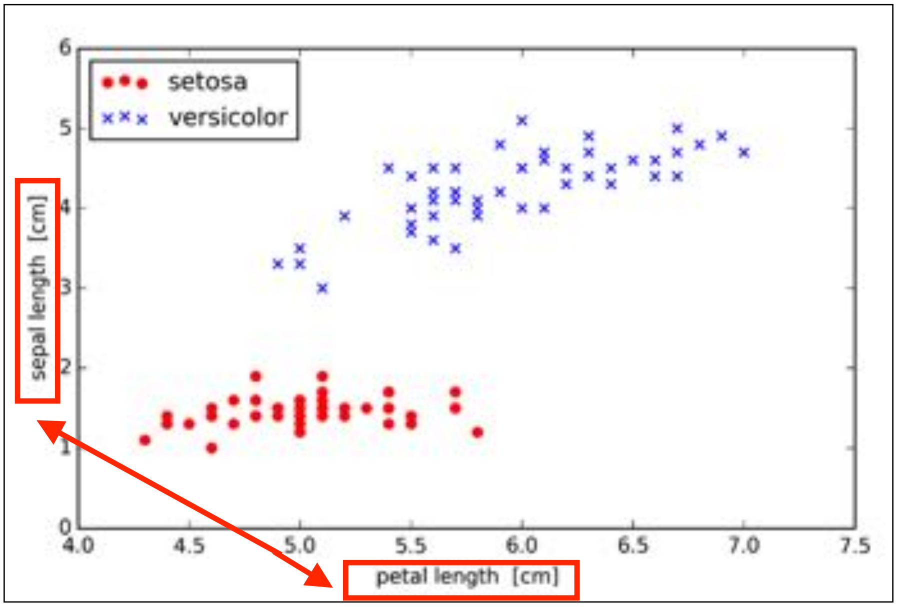
> Instead of using NumPy to calculate the vector dot product between two arrays a and b via `a.dot(b)` or `np.dot(a, b)`, we could also perform the calculation in pure Python via sum([~~j~~i*j for i,j in zip(a, b)]. However, the advantage of using NumPy over classic Python for-loop structures is that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

Training a perceptron model on the Iris dataset

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

```
>>> 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_,
```

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

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

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

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

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_{\underset{k}{\cancel{i}}} \left(w_{\underset{k}{\cancel{j}}}^{\cancel{(i)}} x_{\underset{k}{\cancel{j}}}^{(i)} \right) \right)$$

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

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

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

Implementing an Adaptive Linear Neuron in Python

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

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

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

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

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

    # highlight test samples
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                alpha=1.0, ~~linewidth~~=1, marker='o',
                s=55, label='test set')
```

Although “linewidth=1” works, it would be better to use “linewidths=1” (with an “s”)

With the slight modification that we made to the `plot_decision_regions` function (highlighted in the preceding code), we can now specify the indices of the samples that we want to mark on the resulting plots. The code is as follows:

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

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

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

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

    # highlight test samples
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                alpha=1.0, linewidth=1, marker='o',
                s=55, label='test set')
```

s

It should be “linewidths” with an “s” instead of “linewidth”

With the slight modification that we made to the `plot_decision_regions` function (highlighted in the preceding code), we can now specify the indices of the samples that we want to mark on the resulting plots. The code is as follows:

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

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:

$$logit\big(p(y=1\,|\,\boldsymbol{x})\big) = w_{\mathbf{0}}x_{\mathbf{0}} + w_{\mathbf{1}}x_{\mathbf{1}} + \cdots + w_m x_m = \sum_{i=0}^{\cancel{n}\ m} \underset{i}{w_{\cancel{m}}}\,\underset{i}{x_{\cancel{m}}} = \boldsymbol{w}^T\boldsymbol{x}$$

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

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

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

Here, z is the net input, that is, the linear combination of weights and sample features and can be calculated as $z = \boldsymbol{w}^T\boldsymbol{x} = w_0 + w_{\mathbf{1}}x_{\mathbf{1}} + \cdots + w_{\boldsymbol{m}}x_{\boldsymbol{m}}$.

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:

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

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

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

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

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

We define $\Delta \boldsymbol{w}$ as follows:

$$\Delta \boldsymbol{w} = \eta \nabla l\left(\boldsymbol{w}\right)$$

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

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)} \cancel{)}$$

x should be outside the parenthese
like in the equation on top of that
page

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}, \; \Delta \boldsymbol{w} = -\eta \nabla J\left(\boldsymbol{w}\right)$$

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

Tackling overfitting via regularization

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

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”

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

Via tl ning
data the
regul

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

or

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

The p
scikit
the tc
whicl λ,

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”

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

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

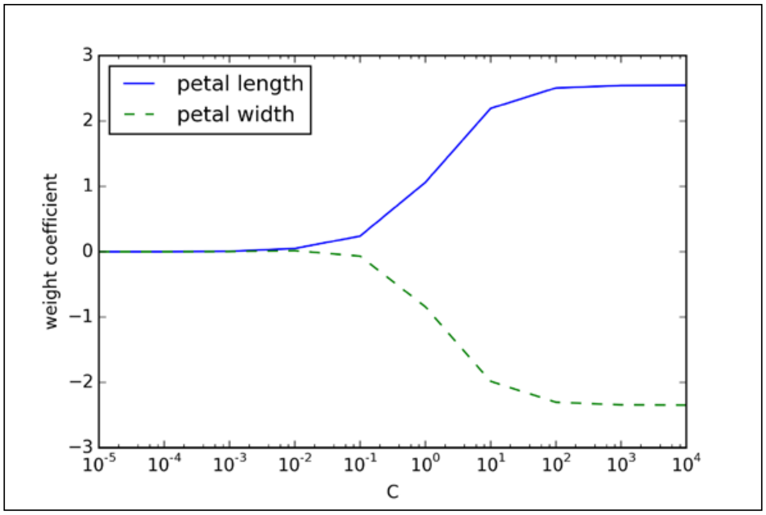
like above, also consider this alt. form for readability

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

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

By executing the preceding code, we fitted ten logistic regression models with different values for the inverse-regularization parameter `C`. For the purposes of illustration, we only collected the weight coefficients of the class 2 vs. all classifier. Remember that we are using the OvR technique for multiclass classification.

As we can see in the resulting plot, the weight coefficients shrink if we decrease the parameter **C**, that is, if we increase the regularization strength:



The positive-values slack variable is simply added to the linear constraints:

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

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

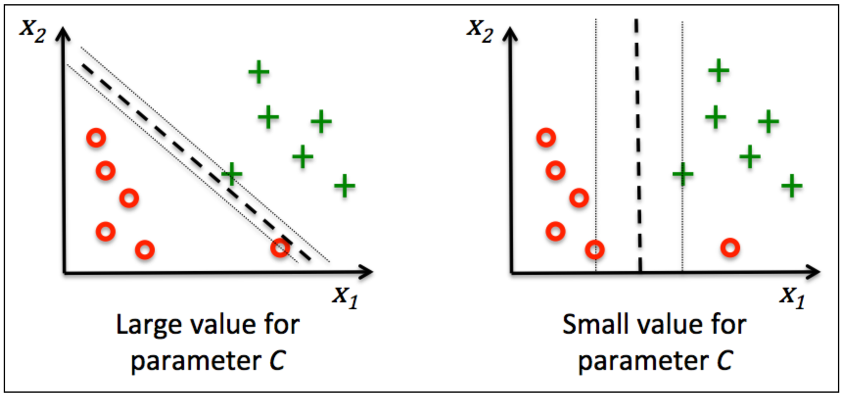
Should be:

wT x(i) ≥1 − ξ (i) if y(i) = 1
wT x(i) ≤ −1 + ξ (i) if y(i) = -1

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

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

Using the variable `C`, we can then control the penalty for misclassification. Large values of `C` correspond to large error penalties whereas we are less strict about misclassification errors if we choose smaller values for `C`. We can then we use the parameter `C` to control the width of the margin and therefore tune the bias-variance trade-off as illustrated in the following figure:



This concept is related to regularization, which we discussed previously in the context of regularized regression where increasing the value of `C` increases the bias and lowers the variance of the model.

It should be: “increasing the value of lambda increases the bias …” (lambda instead of C) or “decreasing the value of C increases the bias”

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

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

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 ($p\left(i|t\right) \neq 0$):

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

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

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

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

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

$$I_G(t)= 1-\sum_{i=1}^{c} 0.5^2 = 0.5$$

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:

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

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

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

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

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

$$A: \overset{\text{IG\_G}}{\cancel{I}_G} = 0.5 - \frac{4}{8}0.375 - \frac{4}{8}0.375 = 0.125$$

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

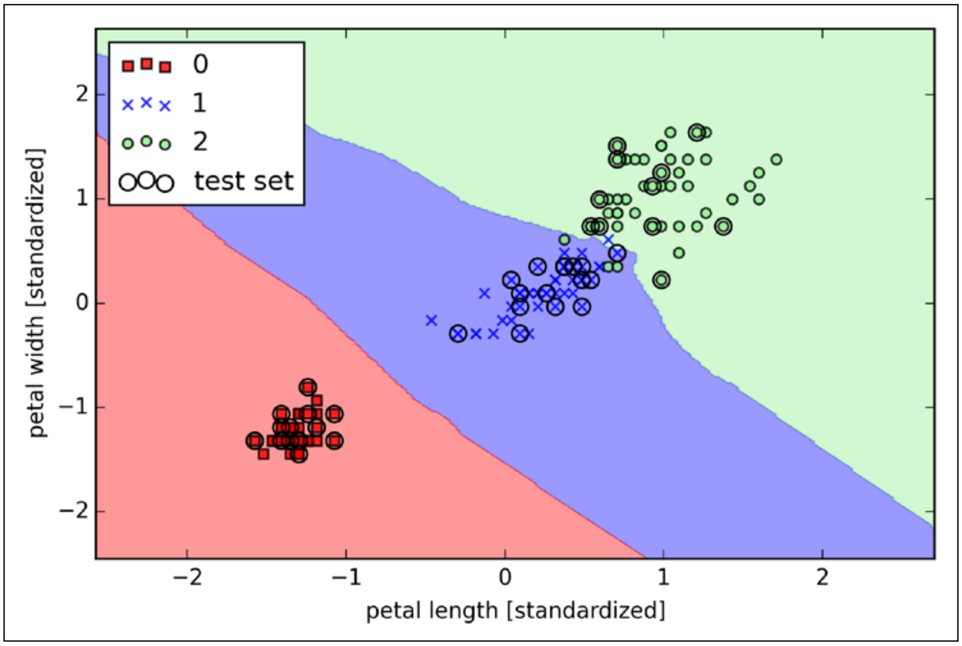
$$B: I_G\left(D_{right}\right) = 1 - \left(1^2 + 0^2\right) = 0$$

$$B: IG_G = 0.5 - \frac{6}{8}0.\overline{4} - 0 = 0.\overline{16}$$

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

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

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



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:

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

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

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

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

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

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

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

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

Eliminating samples or features with missing values

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

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

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

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

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

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

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

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

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

Imputing missing values

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

```
>>> from sklearn.preprocessing import Imputer
>>> imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imr = imr.fit(df)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,7.5~~3.~~,   8.],
       [ 10.,  11.,  12., 6~~4.~~]])
```

Here, we replaced each `NaN` value by the corresponding mean, which is separately calculated for each feature column. If we changed the setting `axis=0` to `axis=1`, we'd calculate the row means. Other options for the `strategy` parameter are `median` or `most_frequent`, where the latter replaces the missing values by the most frequent values. This is useful for imputing categorical feature values.

Understanding the scikit-learn estimator API

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

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

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

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

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

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

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

standardized
-1.46385
-0.87831
-0.29277
0.29277
0.87831
1.46385

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

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

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

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

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

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

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

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

The weight array that we accessed via the `lr.coef_` attribute contains three rows of weight coefficients, one weight vector for each class. Each row consists of 13 weights where each weight is multiplied by the respective feature in the 13-dimensional Wine dataset to calculate the net input:

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

To include the bias unit, the “1”s should
be changed to a 0 (like in chapter 2). However, please note that
scikit learn stores the bias and the weights
separately; so, it’s maybe better to write

z = w_{1}x_{1} + … w_{m}x_{m} + b = \sum^{m}_j=1 x_{j}w_{j} + b = w^{T} x + b

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

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

1. Initialize the algorithm with $k = d$, where d is the dimensionality of the full feature space $\mathbf{X}_d$.
2. Determine the feature x^- that maximizes the criterion $\boldsymbol{x}^- = argmaxJ\left(\boldsymbol{X}_k - \boldsymbol{x}\right)$ where $\boldsymbol{x} \in \boldsymbol{X}_k$.
3. $X_{k-1} := X_k - x^- ;\ k := k - 1$
   Remove the feature x^- from the feature set: ~~$\boldsymbol{X}_k - 1 = \boldsymbol{X}_k - 1 = \boldsymbol{X}_k - \boldsymbol{x}^-, k = k - 1$~~.
4. Terminate if k equals the number of desired features, if not, go to step 2.

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

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

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

Assessing feature importance with random forests

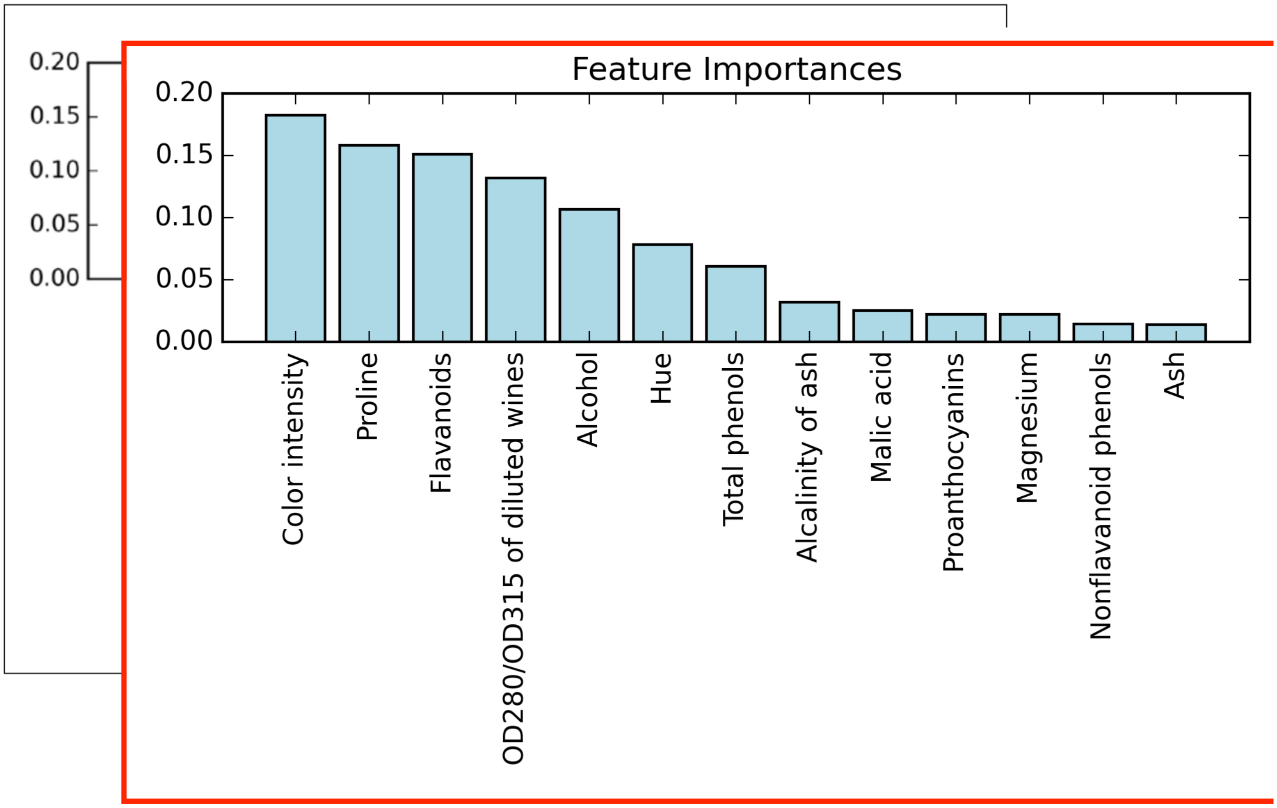
In the previous sections, you learned how to use L1 regularization to zero out irrelevant features via logistic regression and use the SBS algorithm for feature selection. Another useful approach to select relevant features from a dataset is to use a random forest, an ensemble technique that we introduced 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],~~   feat_labels[indices[f]],
...                             importances[indices[f]]))
~~1) Alcohol                        0.182508~~
~~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)     feat_labels[indices]
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()
```

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



We can conclude that the Color intensity ~~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: Color intensity, Proline, and Flavonoids

```
>>> 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.

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)
```

This should be just “transform”, not “fit_transform”

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.

Now, let's obtain the eigenpairs of the covariance matrix. As we surely remember from our introductory linear algebra or calculus classes, an eigen~~value~~ vector $\boldsymbol{v}$ 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`).

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

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

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

Feature transformation

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

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

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

There’s a missing whitespace between “in” and “range”

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

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

4. Compute the eigenvectors and corresponding eigenvalues of the matrix $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$.
5. Choose the k eigenvectors that correspond to the k largest eigenvalues to construct a $d \times k$-dimensional transformation matrix $\boldsymbol{W}$; the eigenvectors are the columns of this matrix.
6. Project the samples onto the new feature subspace using the transformation matrix $\boldsymbol{W}$.

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

Computing the scatter matrices

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

$$\boldsymbol{m}_i = \frac{1}{n_i} \sum_{\boldsymbol{x} \in D_i}^{c} \boldsymbol{x}_m$$

This results in three mean vectors:

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{i,\, alcohol} \\ \mu_{i,\, malic\ acid} \\ \vdots \\ \mu_{i,\, proline} \end{bmatrix}^{T} \quad i \in \{1,2,3\}$$

Note the “T” for “transpose” above. Although, NumPy would handle this case, it would be mathematically wrong to subtract a column vector (m_i) from row vectors (samples). I remember that I displayed the mean vectors as a column vector for visual purposes since the row-vector representation looked a bit ugly. Somehow, the superscript “T” must have gone missing during the layout stage.tor in the later sections

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

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

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

Using the mean vectors, we can now compute the within-class scatter matrix $\boldsymbol{S}_W$:

$$\boldsymbol{S}_W = \sum_{i=1}^{c} \boldsymbol{S}_i$$

This is calculated by summing up the individual scatter matrices $\boldsymbol{S}_i$ of each individual class i:

$$\boldsymbol{S}_i = \sum_{\boldsymbol{x} \in D_i}^{c} \left(\boldsymbol{x} - \boldsymbol{m}_i\right)\left(\boldsymbol{x} - \boldsymbol{m}_i\right)^T$$

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

Should be “X_train” instead of “X”
also, it should be “y_train” instead of “y”

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

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

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

$$\Sigma_i = \frac{1}{N_i}\boldsymbol{S}_W = \frac{1}{N_i}\sum_{\boldsymbol{x}\in D_i}^{c}\left(\boldsymbol{x}-\boldsymbol{m}_i\right)\left(\boldsymbol{x}-\boldsymbol{m}_i\right)^T$$

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

After we have computed the scaled within-class scatter matrix (or covariance matrix), we can move on to the next step and compute the between-class scatter matrix $\boldsymbol{S}_B$:

$$\boldsymbol{S}_B = \sum_{i=1}^{c} N_i\left(\boldsymbol{m}_i-\boldsymbol{m}\right)\left(\boldsymbol{m}_i-\boldsymbol{m}\right)^T$$

Here, $\boldsymbol{m}$ is the overall mean that is computed, including samples from all classes.

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

Should be “X_train” instead of “X” and “y_train” instead of “y’

The three dots must have gone lost during the layout; the S_B should be within the for-loop of course!

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

Selecting linear discriminants for the new feature subspace

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

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

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

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


Eigenvalues in decreasing order:

~~643.015384346~~             452.721581245
~~225.086981854~~             156.43636122
~~1.37146633984e-13~~         8.11327596465e-14
~~5.68434188608e-14~~         2.78687384543e-14
~~4.16877714935e-14~~         2.78687384543e-14
~~4.16877714935e-14~~         2.27622032758e-14
~~3.76733516161e-14~~         2.27622032758e-14
~~3.7544790902e-14~~          1.97162599817e-14
~~3.7544790902e-14~~          1.32484714652e-14
~~2.30295239559e-14~~         1.32484714652e-14
~~2.30295239559e-14~~         1.03791501611e-14
~~1.9101018959e-14~~          5.94140664834e-15
~~3.86601693797e-16~~         2.12636975748e-16
```

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

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
 [[-0.0707 -0.3778]
 [ 0.0359 -0.2223]
 [-0.0263 -0.3813]
 [ 0.1875  0.2955]
 [-0.0033  0.0143]
 [ 0.2328  0.0151]
 [-0.7719  0.2149]
 [-0.0803  0.0726]
 [ 0.0896  0.1767]
 [ 0.1815 -0.2909]
 [-0.0631  0.2376]
 [-0.3794  0.0867]
 [-0.3355 -0.586 ]]
```

```
Matrix W:
 [[ 0.0662 -0.3797]
 [-0.0386 -0.2206]
 [ 0.0217 -0.3816]
 [-0.184   0.3018]
 [ 0.0034  0.0141]
 [-0.2326  0.0234]
 [ 0.7747  0.1869]
 [ 0.0811  0.0696]
 [-0.0875  0.1796]
 [-0.185  -0.284 ]
 [ 0.066   0.2349]
 [ 0.3805  0.073 ]
 [ 0.3285 -0.5971]]
```

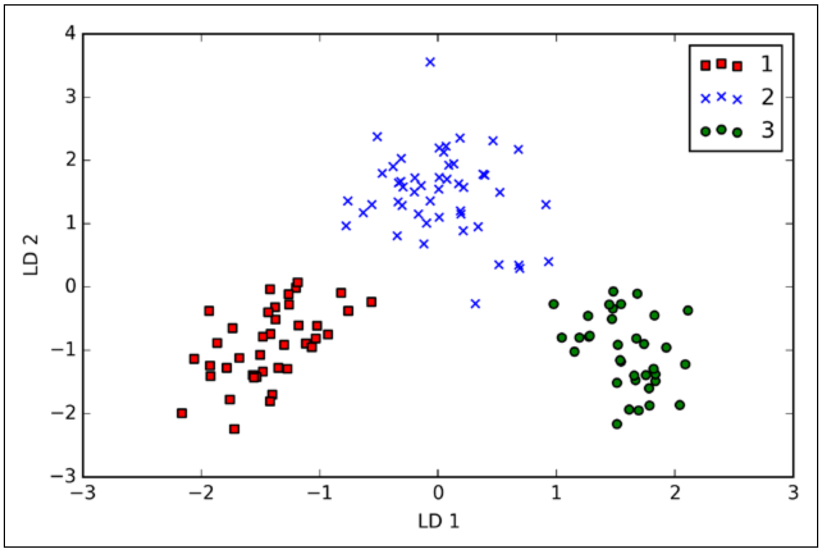
Projecting samples onto the new feature space

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

$$X' = XW$$

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

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



Please see the IPython notebook for an updated figure

LDA via scikit-learn

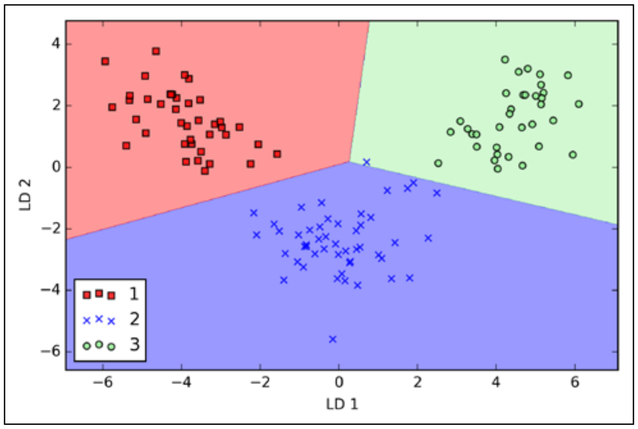
The step-by-step implementation was a good exercise for understanding the inner workings of LDA and understanding the differences between LDA and PCA. Now, let's take a look at the `LDA` class implemented in scikit-learn:

```
>>> from sklearn.lda import LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

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

```
>>> lr = LogisticRegression()
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Looking at the resulting plot, we see that the logistic regression model misclassifies one of the samples from class 2:

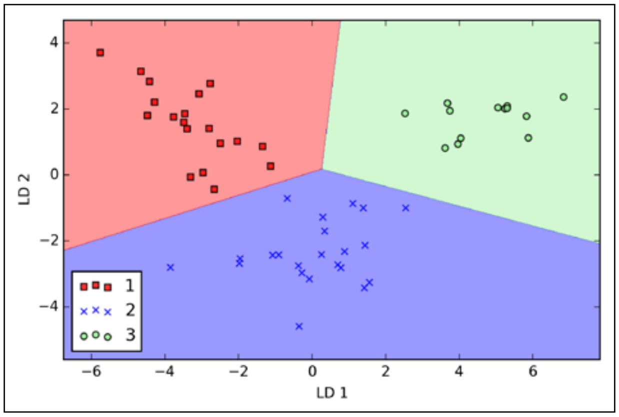


Please see the IPython notebook for an updated figure

By lowering the regularization strength, we could probably shift the decision boundaries so that the logistic regression models classify all samples in the training dataset correctly. However, let's take a look at the results on the test set:

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

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



Please see the IPython notebook for an updated figure

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

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

$$\downarrow \phi$$

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

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

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

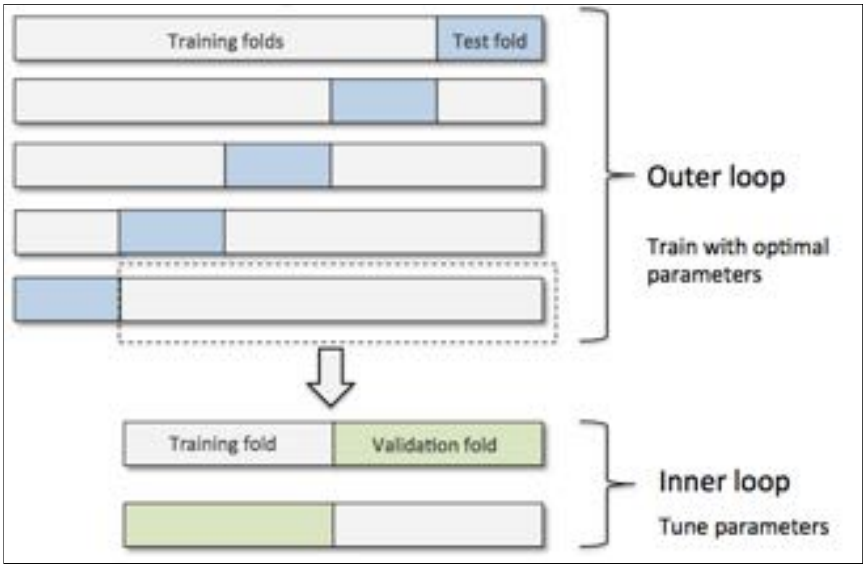
$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^{n} \left(x_j^{(i)} - \mu_j \right) \left(x_k^{(i)} - \mu_k \right)$$

Since the standardizing of features centers them at mean zero, for instance, $\frac{1}{n} \sum_i x_j^{(i)} = 0$, we can simplify this equation as follows:

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

This is of course also true for x_k. Maybe it’s better to write $\mu_j = 0$ and $\mu_k = 0$.

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**:



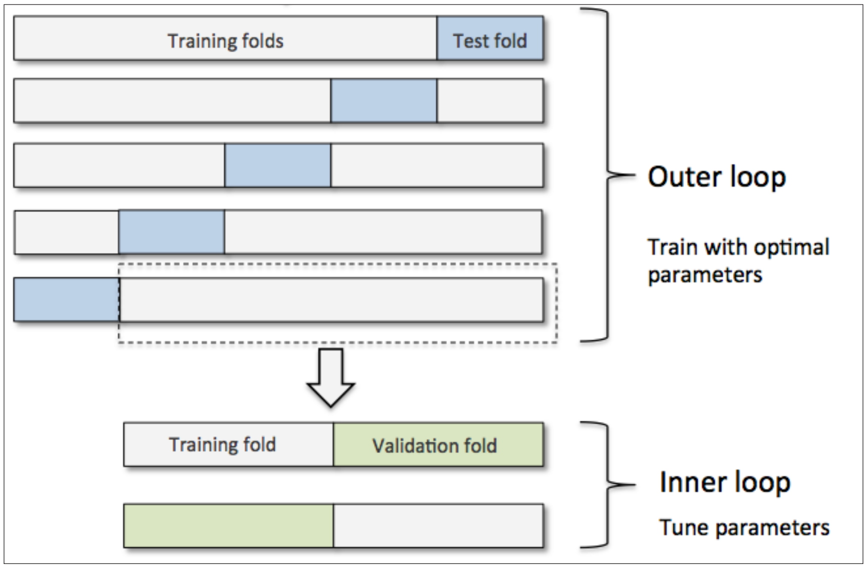
In scikit-learn, we can perform nested cross-validation as follows:

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

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=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
```

[Note that the accuracy score is still correct, the
the error above was a typ n the chapter text template; the
code in the accompanying IPython Notebook.

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



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

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=2,
...                   n_jobs=-1)
>>> scores = cross_val_score(gs, X_train, y_train, scoring='accuracy',
cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...               np.mean(scores), np.std(scores)))
CV accuracy: ~~0.978 +/- 0.012~~
```

0.965 +/- 0.025

The returned average cross-validation accuracy gives us a good estimate of what to expect if we tune the hyperparameters of a model and then use it on unseen data. For example, we can use the nested cross-validation approach to compare an SVM model to a simple decision tree classifier; for simplicity, we will only tune its depth parameter:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...       estimator=DecisionTreeClassifier(random_state=0),
...       param_grid=[
...            {'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
...       scoring='accuracy',
...       cv=5)
>>> scores = cross_val_score(gs,
...                           X_train,
...                           y_train,
...                           scoring='accuracy',
...                           cv=~~5~~)   2
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                     np.mean(scores), np.std(scores)))
CV accuracy: 0~~.908 +/- 0.045~~  0.921 +/- 0.029
```

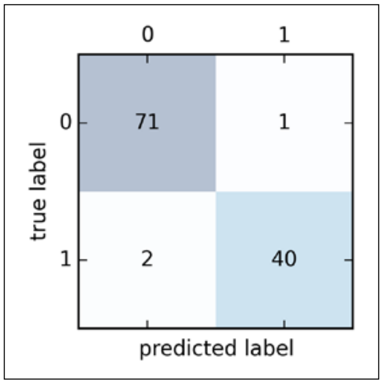
As we can see here, the nested cross-validation performance of the SVM model (97.8 percent) is notably better than the performance of the decision tree (90.8 percent). Thus, we'd expect that it might be the better choice for classifying new data that comes from the same population as this particular dataset.

Looking at different performance evaluation metrics

In the previous sections and chapters, we evaluated our models using the model accuracy, which is a useful metric to quantify the performance of a model in general. However, there are several other performance metrics that can be used to measure a model's relevance, such as **precision**, **recall**, and the **F1-score**.

```
>>> plt.xlabel('predicted label')
>>> plt.ylabel('true label')
>>> plt.show()
```

Now, the confusion matrix plot as shown here should make the results a little bit easier to interpret:



Assuming that class 1 (malignant) is the positive class in this example, our model correctly classified 71 of the samples that belong to class 0 (~~false~~ true negatives) and 40 samples that belong to class 1 (true positives), respectively. ~~However, our model also incorrectly misclassified 2 samples from class 0 as class 1 (false negatives), and it predicted that 1 sample is benign although it is a malignant tumor (false positive)~~. In the next section, we will learn how we can use this information to calculate various different error metrics.

However, our model also incorrectly misclassified 1 sample from class 0 as class 1 (false positive), and it predicted that 2 samples are benign although it is a malignant tumor (false negatives).

Optimizing the precision and recall of a classification model

Both the prediction **error** (**ERR**) and **accuracy** (**ACC**) provide general information about how many samples are misclassified. The error can be understood as the sum of all false predictions divided by the number of total predications, and the accuracy is calculated as the sum of correct predictions divided by the total number of predictions, respectively:

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

> Similar to ROC curves, we can compute **precision-recall curves** for the different probability thresholds of a classifier. A function for plotting those precision-recall curves is also implemented in scikit-learn and is documented at `http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html`.

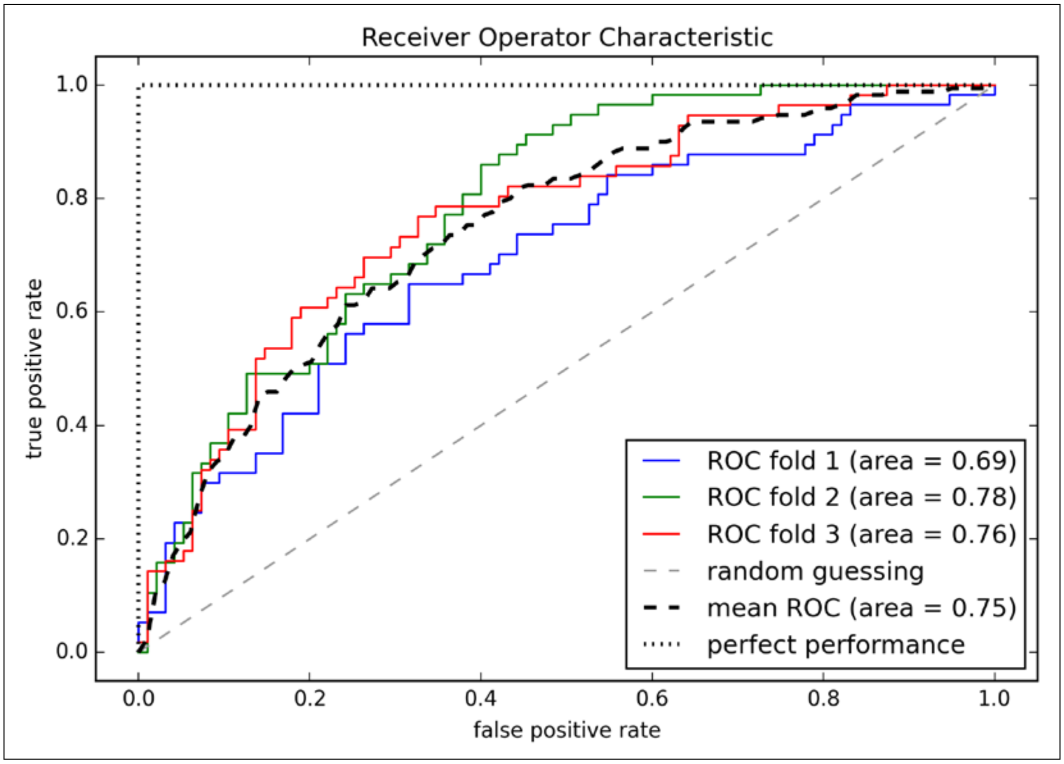
By executing the following code example, we will plot an ROC curve of a classifier that only uses two features from the Breast Cancer Wisconsin dataset to predict whether a tumor is benign or malignant. Although we are going to use the same logistic regression pipeline that we defined previously, we are making the classification task more challenging for the classifier so that the resulting ROC curve becomes visually more interesting. For similar reasons, we are also reducing the number of folds in the `StratifiedKFold` validator to three. The code is as follows:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp
>>> X_train2 = X_train[:, [4, 14]]
>>> cv = StratifiedKFold(y_train,
...                      n_folds=3,
...                      random_state=1)
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []

>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(X_train2[train],
>>> y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
```

ease insert these
nes here

```
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...                     ('pca', PCA(n_components=2)),
...                     ('clf', LogisticRegression(penalty='l2',
...                                                random_state=0,
...                                                C=100.0))])
```

In the preceding code example, we used the already familiar `StratifiedKFold` class from scikit-learn and calculated the ROC performance of the `LogisticRegression` classifier in our `pipe_lr` pipeline using the `roc_curve` function from the `sklearn.metrics` module separately for each iteration. Furthermore, we interpolated the average ROC curve from the three folds via the `interp` function that we imported from SciPy and calculated the area under the curve via the `auc` function. The resulting ROC curve indicates that there is a certain degree of variance between the different folds, and the average ROC AUC (0.75) falls between a perfect score (1.0) and random guessing (0.5):



If we are just interested in the ROC AUC score, we could also directly import the `roc_auc_score` function from the `sklearn.metrics` submodule. The following code calculates the classifier's ROC AUC score on the independent test dataset after fitting it on the two-feature training set:

```
>>> ~~pipe_svc~~ = ~~pipe_svc~~.fit(X_train2, y_train)
>>> y_pred2 = ~~pipe_svc~~.predict(X_test[:, [4, 14]])
```

ll "pipe_svc" should be
eplaced by "pipe_lr"

```
>>> from sklearn.metrics import roc_auc_score
>>> from sklearn.metrics import accuracy_score
>>> print('ROC AUC: %.3f' % roc_auc_score(
...         y_true=y_test, y_score=y_pred2))
ROC AUC: ~~0.671~~   0.662

>>> print('Accuracy: %.3f' % accuracy_score(
...         y_true=y_test, y_pred=y_pred2))
Accuracy: ~~0.728~~  0.711
```

Reporting the performance of a classifier as the ROC AUC can yield further insights in a classifier's performance with respect to imbalanced samples. However, while the accuracy score can be interpreted as a single cut-off point on a ROC curve, A. P. Bradley showed that the ROC AUC and accuracy metrics mostly agree with each other (A. P. Bradley. *The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms*. Pattern recognition, 30(7):1145–1159, 1997).

The scoring metrics for multiclass classification

The scoring metrics that we discussed in this section are specific to binary classification systems. However, scikit-learn also implements **macro** and **micro** averaging methods to extend those scoring metrics to multiclass problems via **One vs. All** (**OvA**) classification. The micro-average is calculated from the individual true positives, true negatives, false positives, and false negatives of the system. For example, the micro-average of the precision score in a k-class system can be calculated as follows:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

The macro-average is simply calculated as the average scores of the different systems:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Micro-averaging is useful if we want to weight each instance or prediction equally, whereas macro-averaging weights all classes equally to evaluate the overall performance of a classifier with regard to the most frequent class labels.

If we are using binary performance metrics to evaluate multiclass classification models in scikit-learn, a normalized or weighted variant of the macro-average is used by default. The weighted macro-average is calculated by weighting the score of each class label by the number of true instances when calculating the average. The weighted macro-average is useful if we are dealing with class imbalances, that is, different numbers of instances for each label.

While the weighted macro-average is the default for multiclass problems in scikit-learn, we can specify the averaging method via the `average` parameter inside the different scoring functions that we import from the ~~`sklean`~~`.metrics` module, for example, the `precision_score` or `make_scorer` functions:

“sklearn” instead of “sklean”

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                          pos_label=1,
...                          greater_is_better=True,
...                          average='micro')
```

Summary

In the beginning of this chapter, we discussed how to chain different transformation techniques and classifiers in convenient model pipelines that helped us to train and evaluate machine learning models more efficiently. We then used those pipelines to perform k-fold cross-validation, one of the essential techniques for model selection and evaluation. Using k-fold cross-validation, we plotted learning and validation curves to diagnose the common problems of learning algorithms, such as overfitting and underfitting. Using grid search, we further fine-tuned our model. We concluded this chapter by looking at a confusion matrix and various different performance metrics that can be useful to further optimize a model's performance for a specific problem task. Now, we should be well-equipped with the essential techniques to build supervised machine learning models for classification successfully.

In the next chapter, we will take a look at ensemble methods, methods that allow us to combine multiple models and classification algorithms to boost the predictive performance of a machine learning system even further.

To see bagging in action, let's create a more complex classification problem using the **Wine** dataset that we introduced in *Chapter 4, Building Good Training Sets – Data Preprocessing*. Here, we will only consider the Wine classes 2 and 3, and we select two features: **Alcohol** and **Hue**.

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash',
...                    'Magnesium', 'Total phenols',
...                    'Flavanoids', 'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> df_wine = df_wine[df_wine['Class label'] != 1]
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol', 'Hue']].values
```

Next we encode the class labels into binary format and split the dataset into 60 percent training and 40 percent test set, respectively:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.cross_validation import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test =\
...            train_test_split(X, y,
...                             test_size=0.40,
...                             random_state=1)
```

A `BaggingClassifier` algorithm is already implemented in scikit-learn, which we can import from the `ensemble` submodule. Here, we will use an unpruned decision tree as the base classifier and create an ensemble of 500 decision trees fitted on different bootstrap samples of the training dataset:

I forgot a
“random_state=1”
here, this should be
like shown below:

```
>>> from sklearn.ensemble import BaggingClassifier
~~>>> tree = DecisionTreeClassifier(criterion='entropy',~~
~~...                                 max_depth=None)~~
>>> bag = BaggingClassifier(base_estimator=tree,
```

```
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=None,
...                               random_state=1)
...                         n_estimators=500,
...                         max_samples=1.0,
...                         max_features=1.0,
...                         bootstrap=True,
...                         bootstrap_features=False,
...                         n_jobs=1,
...                         random_state=1)
```

Next we will calculate the accuracy score of the prediction on the training and test dataset to compare the performance of the bagging classifier to the performance of a single unpruned decision tree:

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...        % (tree_train, tree_test))
Decision tree train/test accuracies 1.000/~~0.854~~ 0.833
```

Based on the accuracy values that we printed by executing the preceding code snippet, the unpruned decision tree predicts all class labels of the training samples correctly; however, the substantially lower test accuracy indicates high variance (overfitting) of the model:

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Bagging train/test accuracies %.3f/%.3f'
...        % (bag_train, bag_test))
Bagging train/test accuracies 1.000/0.896
```

Although the training accuracies of the decision tree and bagging classifier are similar on the training set (both 1.0), we can see that the bagging classifier has a slightly better generalization performance as estimated on the test set. Next let's compare the decision regions between the decision tree and bagging classifier:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
```

To walk through the AdaBoost illustration step by step, we start with subfigure **1**, which represents a training set for binary classification where all training samples are assigned equal weights. Based on this training set, we train a decision stump (shown as a dashed line) that tries to classify the samples of the two classes (triangles and circles) as well as possible by minimizing the cost function (or the impurity score in the special case of decision tree ensembles). For the next round (subfigure **2**), we assign a larger weight to the two previously misclassified samples (circles). Furthermore, we lower the weight of the correctly classified samples. The next decision stump will now be more focused on the training samples that have the largest weights, that is, the training samples that are supposedly hard to classify. The weak learner shown in subfigure **2** misclassifies three different samples from the circle-class, which are then assigned a larger weight as shown in subfigure **3**. Assuming that our AdaBoost ensemble only consists of three rounds of boosting, we would then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote, as shown in subfigure **4**.

Now that have a better understanding behind the basic concept of AdaBoost, let's take a more detailed look at the algorithm using pseudo code. For clarity, we will denote element-wise multiplication by the cross symbol $(\times)$ and the dot product between two vectors by a dot symbol $(\cdot)$, respectively. The steps are as follows:

1. Set weight vector $\boldsymbol{w}$ to uniform weights where $\sum_i w_i = 1$
2. For *j* in *m* boosting rounds, do the following:
3. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
4. Predict class labels: $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$.
5. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} == \boldsymbol{y})$.
6. Compute coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
7. Update weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
8. Normalize weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
9. Compute final prediction: $\hat{\boldsymbol{y}} = \left(\sum_{j=1}^{m} \left(\boldsymbol{\alpha}_j \times \text{predict}(C_j, \boldsymbol{X})\right) > 0\right)$.

Note that the expression $(\hat{\boldsymbol{y}} == \boldsymbol{y})$ in step 5 refers to a vector of 1s and 0s, where a 1 is assigned if the prediction is ~~correct~~ and 0 is assigned otherwise.

“incorrect” instead of “correct”

After we have computed the coefficient α_j we can now update the weight vector using the following equation:

$$\boldsymbol{w} := \boldsymbol{w} \times \exp\left(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y}\right)$$

Here, $\hat{\boldsymbol{y}} \times \boldsymbol{y}$ is an element-wise multiplication between the vectors of the predicted and true class labels, respectively. Thus, if a prediction $\hat{y}_i$ is correct, $\hat{y}_i \times y_i$ will have a positive sign so that we decrease the *ith* weight since α_j is a positive number as well:

$0.1 \times \exp\left(-0.424 \times 1 \times 1\right) \approx$ ~~0.066~~ 0.065

Similarly, we will ~~downweight~~ increase the *ith* weight if $\hat{y}_i$ predicted the label incorrectly like this:

$$0.1 \times \exp\left(-0.424 \times 1 \times \left(-1\right)\right) \approx 0.153$$

Or like this:

$$0.1 \times \exp\left(-0.424 \times \left(-1\right) \times \left(1\right)\right) \approx 0.153$$

After we update each weight in the weight vector, we normalize the weights so that they sum up to 1 (step 8):

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i w_i}$$

Here, $\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$.

Thus, each weight that corresponds to a correctly classified sample will be reduced from the initial value of 0.1 to ~~$0.066 / 0.914 \approx 0.072$~~ for the next round 0.065/0.914 ≈ 0.071 of boosting. Similarly, the weights of each incorrectly classified sample will increase from 0.1 to $0.153 / 0.914 \approx 0.167$.

This was AdaBoost in a nutshell. Skipping to the more practical part, let's now train an AdaBoost ensemble classifier via scikit-learn. We will use the same Wine subset that we used in the previous section to train the bagging meta-classifier. Via the `base_estimator` attribute, we will train the `AdaBoostClassifier` on 500 decision tree stumps:

I forgot a "random_state=0" here, this should be like shown below:

```
>>> from sklearn.ensemble import AdaBoostClassifier
~~>>> tree = DecisionTreeClassifier(criterion='entropy'~~,
~~...                                max_depth=1)~~
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=0)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.845/0.854
```

```
>>> tree = DecisionTreeClassifier(criterion='entrop
...                               max_depth=None,
...                               random_state=0)
```

As we can see, the decision tree stump ~~seems to overfit~~ tends to underfit the training data in contrast with the unpruned decision tree that we saw in the previous section:

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('AdaBoost train/test accuracies %.3f/%.3f'
...       % (ada_train, ada_test))
AdaBoost train/test accuracies 1.000/0.875
```

As we can see, the AdaBoost model predicts all class labels of the training set correctly and also shows a slightly improved test set performance compared to the decision tree stump. However, we also see that we introduced additional variance by our attempt to reduce the model bias.

In this chapter, we will be working with a large dataset of movie reviews from the **Internet Movie Database** (**IMDb**) that has been collected by Maas et al. (A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. *Learning Word Vectors for Sentiment Analysis*. In the proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics). The movie review dataset consists of 50,000 polar movie reviews that are labeled as either *positive* or *negative*; here, positive means that a movie was rated with more than six stars on IMDb, and negative means that a movie was rated with fewer than five stars on IMDb. In the following sections, we will learn how to extract meaningful information from a subset of these movie reviews to build a machine learning model that can predict whether a certain reviewer liked or disliked a movie.

A compressed archive of the movie review dataset (84.1 MB) can be downloaded from `http://ai.stanford.edu/~amaas/data/sentiment/` as a gzip-compressed tarball archive:

- If you are working with Linux or Mac OS X, you can open a new terminal window, use `cd` to go into the `download` directory, and execute `tar -zxf aclImdb_v1.tar.gz` to decompress the dataset
- If you are working with Windows, you can download a free archiver such as 7-Zip (`http://www.7-zip.org`) to extract the files from the download archive

Having successfully extracted the dataset, we will now assemble the individual text documents from the decompressed download archive into a single CSV file. In the following code section, we will be reading the movie reviews into a pandas `DataFrame` object, which can take up to 10 minutes on a standard desktop computer. To visualize the progress and estimated time until completion, we will use the **PyPrind** (Python Progress Indicator, `https://pypi.python.org/pypi/PyPrind/`) package that I developed several years ago for such purposes. PyPrind can be installed by executing the command: `pip install pyprind`.

```
>>> import pyprind
>>> import pandas as pd
>>> import os
>>> pbar = pyprind.ProgBar(50000)
>>> labels = {'pos':1, 'neg':0}
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path ='./aclImdb/%s/%s' % (s, l)
...         for file in os.listdir(path):
~~...             with open(os.path.join(path, file), 'r') as infile:~~
...        with open(os.path.join(path, file), ‘r’, encoding=‘utf-8’) as infile:
```

As we saw in the previous subsection, the word `is` had the largest term frequency in the 3rd document, being the most frequently occurring word. However, after transforming the same feature vector into tf-idfs, we see that the word `is` is now associated with a relatively small tf-idf ~~(`0.31`)~~ in document 3 since it is also contained in documents 1 and 2 and thus is unlikely to contain any useful, discriminatory information.

should be 0.48, like we manually computed it on the following page

However, if we'd manually calculated the tf-idfs of the individual terms in our feature vectors, we'd have noticed that the `TfidfTransformer` calculates the tf-idfs slightly differently compared to the *standard* textbook equations that we defined earlier. The equations for the idf and tf-idf that were implemented in scikit-learn are:

$$\text{idf}\left(\text{t,d}\right) = log\frac{1+n_d}{1+\text{df}\left(\text{d,t}\right)}$$

The tf-idf equation that was implemented in scikit-learn is as follows:

$$\text{tf-idf}\left(\text{t,d}\right) = tf\left(\text{t,d}\right)\times\left(\text{idf}\left(\text{t,d}\right)+1\right)$$

While it is also more typical to normalize the raw term frequencies before calculating the tf-idfs, the `TfidfTransformer` normalizes the tf-idfs directly. By default (`norm='l2'`), scikit-learn's `TfidfTransformer` applies the L2-normalization, which returns a vector of length 1 by dividing an un-normalized feature vector *v* by its L2-norm:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2+v_2^2+\cdots+v_n^2}} = \frac{v}{\left(\sum_{i=1}^{n} v_i^2\right)^{1/2}}$$

To make sure that we understand how `TfidfTransformer` works, let us walk through an example and calculate the tf-idf of the word `is` in the 3rd document.

The word `is` has a term frequency of 2 (tf = 2) in document 3, and the document frequency of this term is 3 since the term `is` occurs in all three documents (df = 3). Thus, we can calculate the idf as follows:

$$idf\left(\text{"is"},\text{d3}\right) = \log\frac{1+3}{1+3} = 0$$

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

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

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):
```

```
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)
...     text = re.sub('[\W]+', ' ', text.lower()) + \
                '.join(emoticons).replace('-', '')
...     return text
```

Via the first regex `<[^>]*>` in the preceding code section, we tried to remove the entire HTML markup that was contained in the movie reviews. Although many programmers generally advise against the use of regex to parse HTML, this regex should be sufficient to *clean* this particular dataset. After we removed the HTML markup, we used a slightly more complex regex to find emoticons, which we temporarily stored as `emoticons`. Next we removed all non-word characters from the text via the regex `[\W]+`, converted the text into lowercase characters, and eventually added the temporarily stored `emoticons` to the end of the processed document string. Additionally, we removed the *nose* character (-) from the emoticons for consistency.

> Although regular expressions offer an efficient and convenient approach to searching for characters in a string, they also come with a steep learning curve. Unfortunately, an in-depth discussion of regular expressions is beyond the scope of this book. However, you can find a great tutorial on the Google Developers portal at `https://developers.google.com/edu/python/regular-expressions` or check out the official documentation of Python's re module at `https://docs.python.org/3.4/library/re.html`.

Although the addition of the emoticon characters to the end of the cleaned document strings may not look like the most elegant approach, the order of the words doesn't matter in our bag-of-words model if our vocabulary only consists of 1-word tokens. But before we talk more about splitting documents into individual terms, words, or tokens, let us confirm that our preprocessor works correctly:

```
>>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

Lastly, since we will make use of the *cleaned* text data over and over again during the next sections, let us now apply our `preprocessor` function to all movie reviews in our `DataFrame`:

```
>>> df['review'] = df['review'].apply(preprocessor)
```

When we initialized the GridSearchCV object and its parameter grid using the preceding code, we restricted ourselves to a limited number of parameter combinations since the number of feature vectors, as well as the large vocabulary, can make the grid search computationally quite expensive; using a standard Desktop computer, our grid search may take up to 40 minutes to complete.

In the previous code example, we replaced the CountVectorizer and TfidfTransformer from the previous subsection with the TfidfVectorizer, which combines the latter transformer objects. Our param_grid consisted of two parameter dictionaries. In the first dictionary, we used the TfidfVectorizer with its default settings (use_idf=True, smooth_idf=True, and norm='l2') to calculate the tf-idfs; in the second dictionary, we set those parameters to use_idf=False, smooth_idf=False, and norm=None in order to train a model based on raw term frequencies. Furthermore, for the logistic regression classifier itself, we trained models using L2 and L1 regularization via the penalty parameter and compared different regularization strengths by defining a range of values for the inverse-regularization parameter C.

After the grid search has finished, we can print the best parameter set:

```
>>> print('Best parameter set: %s ' % gs_lr_tfidf.best_params_)
Best parameter set: {'clf__C': 10.0, 'vect__stop_words': None,
'clf__penalty': 'l2', 'vect__tokenizer': <function tokenizer at
0x7f6c704948c8>, 'vect__ngram_range': (1, 1)}
```

As we can see here, we obtained the best grid search results using the regular tokenizer without Porter stemming, no stop-word library, and tf-idfs in combination with a logistic regression classifier that uses L2 regularization with the regularization strength C=10.0.

Using the best model from this grid search, let us print the average 5-fold cross-validation accuracy score~~s~~ on the training set and the classification accuracy on the test dataset:

```
>>> print('CV Accuracy: %.3f'
...       % gs_lr_tfidf.best_score_)
CV Accuracy: 0.897
>>> clf = gs_lr_tfidf.best_estimator_
>>> print('Test Accuracy: %.3f'
...     % clf.score(X_test, y_test))
Test Accuracy: 0.899
```

The results reveal that our machine learning model can predict whether a movie review is positive or negative with 90 percent accuracy.

```
...             + ' '.join(emoticons).replace('-', '')
...     tokenized = [w for w in text.split() if w not in stop]
...     return tokenized
```

Next we define a generator function, `stream_docs`, that reads in and returns one document at a time:

```
>>> def stream_docs(path):
...    ~~with open(path, 'r') as csv:~~    with open(path, ‘r’, encoding=‘utf-8’) as csv:
...        next(csv) # skip header
...        for line in csv:
...            text, label = line[:-3], int(line[-2])
...            yield text, label
```

To verify that our `stream_docs` function works correctly, let us read in the first document from the `movie_data.csv` file, which should return a tuple consisting of the review text as well as the corresponding class label:

```
>>> next(stream_docs(path='./movie_data.csv'))
('"In 1974, the teenager Martha Moxley ... ',1)
```

We will now define a function, `get_minibatch`, that will take a document stream from the `stream_docs` function and return a particular number of documents specified by the `size` parameter:

```
>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...         try:
...             for _ in range(size):
...                 text, label = next(doc_stream)
...                 docs.append(text)
...                 y.append(label)
...         except StopIteration:
...             return None, None
...         return docs, y
```

Unfortunately, we can't use the `CountVectorizer` for out-of-core learning since it requires holding the complete vocabulary in memory. Also, the `TfidfVectorizer` needs to keep the all feature vectors of the training dataset in memory to calculate the inverse document frequencies. However, another useful vectorizer for text processing implemented in scikit-learn is `HashingVectorizer`. `HashingVectorizer` is data-independent and makes use of the Hashing trick via the 32-bit MurmurHash3 algorithm by Austin Appleby (`https://sites.google.com/site/murmurhash/`).

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier
```

https://developer.mozilla.org/en-US/docs/Web/HTML

Now, let's take a look at the contents of the first_app.html file. If you are not familiar with the HTML syntax yet, I recommend you visit ~~http://www.w3schools.com/html/default.asp~~ for useful tutorials for learning the basics of HTML.

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  </head>
  <body>
  <div>Hi, this is my first Flask web app!</div>
  </body>
</html>
```

Here, we have simply filled an empty HTML template file with a div element (a block level element) that contains the sentence: Hi, this is my first Flask web app!. Conveniently, Flask allows us to run our apps locally, which is useful for developing and testing web applications before we deploy them on a public web server. Now, let's start our web application by executing the command from the terminal inside the 1st_flask_app_1 directory:

```
python3 app.py
```

We should now see a line such as the following displayed in the terminal:

```
* Running on http://127.0.0.1:5000/
```

This line contains the address of our local server. We can now enter this address in our web browser to see the web application in action. If everything has executed correctly, we should now see a simple website with the content: **Hi, this is my first Flask web app!**.

Form validation and rendering

In this subsection, we will extend our simple Flask web application with HTML form elements to learn how to collect data from a user using the **WTForms** library (https://wtforms.readthedocs.org/en/latest/), which can be installed via pip:

```
pip install wtforms
```

```
app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('',[validators.DataRequired()])

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)
```

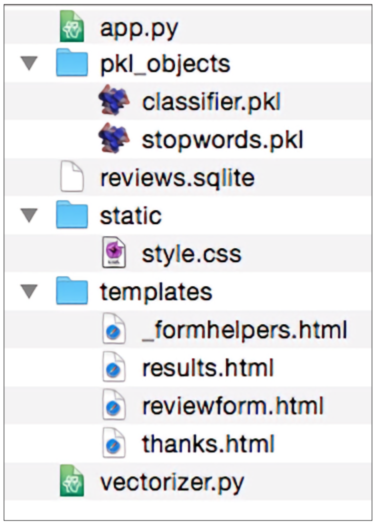
Using wtforms, we extended the index function with a text field that we will embed in our start page using the TextAreaField class, which automatically checks whether a user has provided valid input text or not. Furthermore, we defined a new function, hello, which will render an HTML page hello.html if the form has been validated. Here, we used the POST method to transport the form data to the server in the message body. Finally, by setting the argument debug=True inside the app.run method, we further activated Flask's debugger. This is a useful feature for developing new web applications.

Now, we will implement a generic macro in the file _formhelpers.html via the **Jinja2** templating engine, which we will later import in our first_app.html file to render the text field:

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  ~~</dd>~~
{% endmacro %}
```

</dd>
</dt>

To start with the big picture, let's take a look at the directory tree that we are going to create for this movie classification app, which is shown here:



```
app.py
▼ pkl_objects
    classifier.pkl
    stopwords.pkl
reviews.sqlite
▼ static
    style.css
▼ templates
    _formhelpers.html
    results.html
    reviewform.html
    thanks.html
vectorizer.py
```

In the previous section of this chapter, we already created the `vectorizer.py` file, the SQLite database `reviews.sqlite`, and the `pkl_objects` subdirectory with the pickled Python objects.

The `app.py` file in the main directory is the Python script that contains our Flask code, and we will use the `review.sqlite` database file (which we created earlier in this chapter) to store the movie reviews that are being submitted to our web app. The `templates` subdirectory contains the HTML templates that will be rendered by Flask and displayed in the browser, and the `static` subdirectory will contain a simple CSS file to adjust the look of the rendered HTML code.

a reader
mentioned that
it says "aswell"
instead
of "as well" in
the kindle
version

Since the `app.py` file is rather long, we will conquer it in two steps. The first section of `app.py` imports the Python modules and objects that we are going to need, as well as the code to unpickle and set up our classification model:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np
```

```
# import HashingVectorizer from local dir
from vectorizer import vect

app = Flask(__name__)

######## Preparing the Classifier
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                 'pkl_objects/classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'negative', 1: 'positive'}
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba = ~~np.max(clf.predict_proba(X))~~   clf.predict_proba(X).max()
    return label[y], proba

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO review_db (review, sentiment, date)"\
    " VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()
```

This first part of the `app.py` script should look very familiar to us by now. We simply imported the `HashingVectorizer` and unpickled the logistic regression classifier. Next, we defined a `classify` function to return the predicted class label as well as the corresponding probability prediction of a given text document. The `train` function can be used to update the classifier given that a document and a class label are provided. Using the `sqlite_entry` function, we can store a submitted movie review in our SQLite database along with its class label and timestamp for our personal records. Note that the `clf` object will be reset to its original, pickled state if we restart the web application. At the end of this chapter, you will learn how to use the data that we collect in the SQLite database to update the classifier permanently.

Here, we simply imported the same `_formhelpers.html` template that we defined in the *Form validation and rendering* section earlier in this chapter. The `render_field` function of this macro is used to render a `TextAreaField` where a user can provide a movie review and submit it via the **Submit review** button displayed at the bottom of the page. This `TextAreaField` is 30 columns wide and 10 rows tall.

Our next template, `results.html`, looks a little bit more interesting:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
  <link rel="stylesheet" href="{{ url_for('static',
    filename='style.css') }}">
  </head>
  <body>

<h3>Your movie review:</h3>
<div>{{ content }}</div>

<h3>Prediction:</h3>
<div>This movie review is <strong>{{ prediction }}</strong>
  (probability: {{ probability }}%).</div>

class <div ~~id~~='button'>
  <form action="/thanks" method="post">
    <input type=submit value='Correct' name='feedback_button'>
    <input type=submit value='Incorrect' name='feedback_button'>
    <input type=hidden value='{{ prediction }}' name='prediction'>
    <input type=hidden value='{{ content }}' name='review'>
  </form>
</div>

class <div ~~id~~='button'>
  <form action="/">
    <input type=submit value='Submit another review'>
  </form>
</div>

  </body>
</html>
```

First, we inserted the submitted review as well as the results of the prediction in the corresponding fields `{{ content }}`, `{{ prediction }}`, and `{{ probability }}`. You may notice that we used the `{{ content }}` and `{{ prediction }}` placeholder variables a second time in the form that contains the **Correct** and **Incorrect** buttons. This is a workaround to `POST` those values back to the server to update the classifier and store the review in case the user clicks on one of those two buttons. Furthermore, we imported a CSS file (`style.css`) at the beginning of the `results.html` file. The setup of this file is quite simple; it limits the width of the contents of this web app to 600 pixels and moves the **Incorrect** and **Correct** buttons labeled with the div id `button` down by 20 pixels:

```
body{
  width:600px;
}
.button {   ~~#button{~~
  padding-top: 20px;
}
```

This CSS file is merely a placeholder, so please feel free to adjust it to adjust the look and feel of the web app to your liking.

The last HTML file we will implement for our web application is the `thanks.html` template. As the name suggests, it simply provides a nice *thank you* message to the user after providing feedback via the **Correct** or **Incorrect** button. Furthermore, we put a **Submit another review** button at the bottom of this page, which will redirect the user to the starting page. The contents of the `thanks.html` file are as follows:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
</head>
  <body>

<h3>Thank you for your feedback!</h3>
<div id='button'>
  <form action="/">
    <input type=submit value='Submit another review'>
  </form>
</div>

  </body>
</html>
```

Updating the movie review classifier

While our predictive model is updated on-the-fly whenever a user provides feedback about the classification, the updates to the `clf` object will be reset if the web server crashes or restarts. If we reload the web application, the `clf` object will be reinitialized from the `classifier.pkl` pickle file. One option to apply the updates permanently would be to pickle the `clf` object once again after each update. However, this would become computationally very inefficient with a growing number of users and could corrupt the pickle file if users provide feedback simultaneously. An alternative solution is to update the predictive model from the feedback data that is being collected in the SQLite database. One option would be to download the SQLite database from the PythonAnywhere server, update the `clf` object locally on our computer, and upload the new pickle file to PythonAnywhere. To update the classifier locally on our computer, we create an `update.py` script file in the `movieclassifier` directory with the following contents:

```
import pickle
import sqlite3
import numpy as np
import os

# import HashingVectorizer from local dir
from vectorizer import vect

def update_model(db_path, model, batch_size=10000):

    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute('SELECT * from review_db')

    results = c.fetchmany(batch_size)
    while results:
        data = np.array(results)
        X = data[:, 0]
        y = data[:, 1].astype(int)

        classes = np.array([0, 1])
        X_train = vect.transform(X)
model.partial_fit  ~~clf.partial_fit~~(X_train, y, classes=classes)
        results = c.fetchmany(batch_size)

    conn.close()
return model    ~~return None~~
```

```
    cur_dir = os.path.dirname(__file__)

    clf = pickle.load(open(os.path.join(cur_dir,
                      'pkl_objects',
                      'classifier.pkl'), 'rb'))
    db = os.path.join(cur_dir, 'reviews.sqlite')

clf=update_model    ~~update_model~~(db_path=db, model=clf, batch_size=10000)

    # Uncomment the following lines if you are sure that
    # you want to update your classifier.pkl file
    # permanently.

    # pickle.dump(clf, open(os.path.join(cur_dir,
    #             'pkl_objects', 'classifier.pkl'), 'wb')
    #             , protocol=4)
```

The `update_model` function will fetch entries from the SQLite database in batches of 10,000 entries at a time unless the database contains fewer entries. Alternatively, we could also fetch one entry at a time by using `fetchone` instead of `fetchmany`, which would be computationally very inefficient. Using the alternative `fetchall` method could be a problem if we are working with large datasets that exceed the computer or server's memory capacity.

Now that we have created the `update.py` script, we could also upload it to the `movieclassifier` directory on PythonAnywhere and import the `update_model` function in the main application script `app.py` to update the classifier from the SQLite database every time we restart the web application. In order to do so, we just need to add a line of code to import the `update_model` function from the `update.py` script at the top of `app.py`:

```
# import update function from local dir
from update import update_model
```

We then need to call the `update_model` function in the main application body:

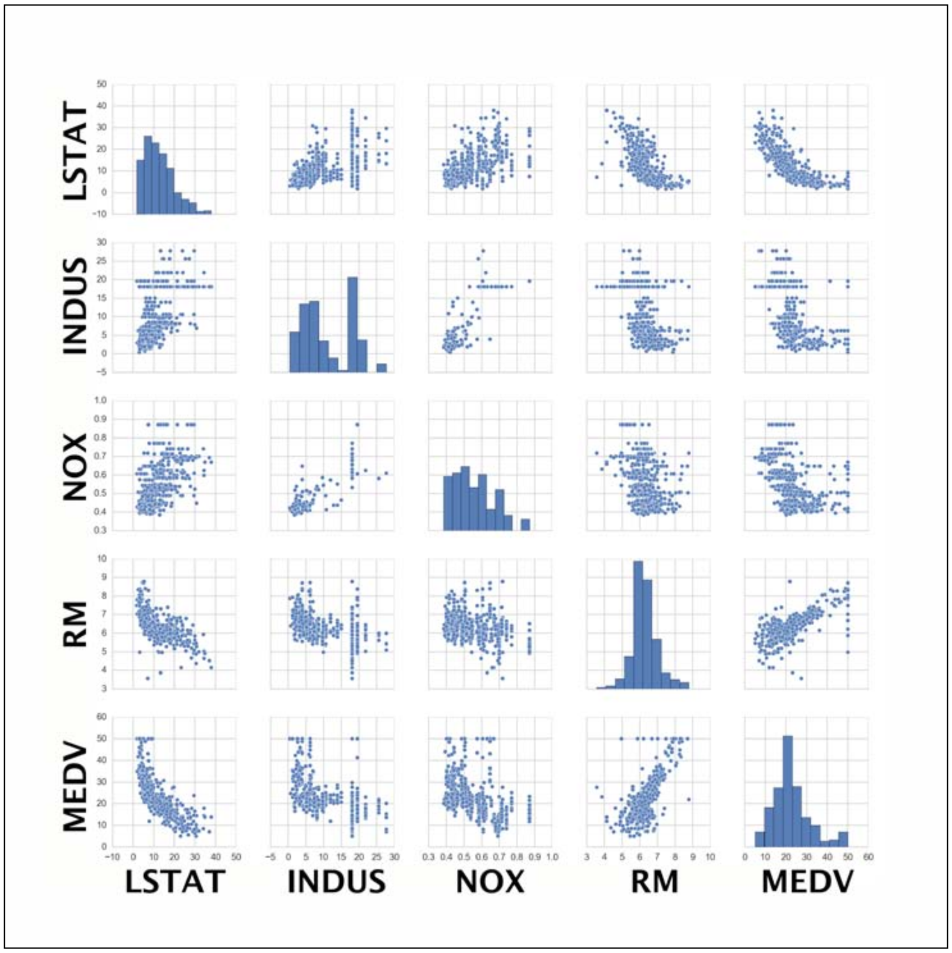
```
…
if __name__ == '__main__':
    ~~update_model(filepath=db~~, model=clf, batch_size=10000)
…
```

clf = update_model(db_path=db

```
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
>>> sns.pairplot(df[cols], size=2.5)~~;~~
>>> plt.show()
```

remove unnecessary semicolon

As we can see in the following figure, the scatterplot matrix provides us with a useful graphical summary of the relationships in a dataset:



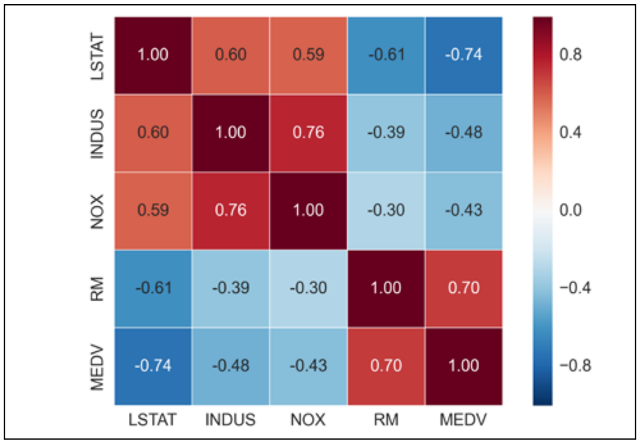
Importing the seaborn library modifies the default aesthetics of matplotlib for the current Python session. If you do not want to use seaborn's style settings, you can reset the matplotlib settings by executing the following command:

```
>>> sns.reset_orig()
```

In the following code example, we will use NumPy's `corrcoef` function on the five feature columns that we previously visualized in the scatterplot matrix, and we will use seaborn's `heatmap` function to plot the correlation matrix array as a heat map:

```
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> sns.set(font_scale=1.5)
>>> hm = sns.heatmap(cm,
...                  cbar=True,
...                  annot=True,
...                  square=True,
...                  fmt='.2f',
...                  annot_kws={'size': 15},
...                  yticklabels=cols,
...                  xticklabels=cols)
>>> plt.show()
```

As we can see in the resulting figure, the correlation matrix provides us with another useful summary graphic that can help us to select features based on their respective linear correlations:



To fit a linear regression model, we are interested in those features that have a high correlation with our target variable **MEDV**. Looking at the preceding correlation matrix, we see that our target variable **MEDV** shows the largest correlation with the **LSTAT** variable (-0.74). However, as you might remember from the scatterplot matrix, there is a clear nonlinear relationship between **LSTAT** and **MEDV**. On the other hand, the correlation between **RM** and **MEDV** is also relatively high (0.70) and given the linear relationship between those two variables that we observed in the scatterplot, **RM** seems to be a good choice for an ~~exploratory~~ variable to introduce the concepts of a simple linear regression model in the following section.

explanatory

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()
```

Delete words “standardized

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

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

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

A Ridge Regression model can be initialized as follows:

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

Note that the regularization strength is regulated `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`.

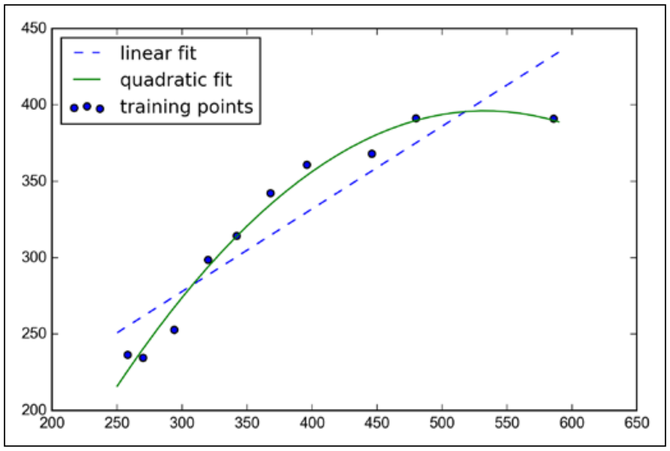
Turning a linear regression model into a curve – polynomial regression

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:

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

delete duplicated "x^2"

In the resulting plot, we can see that the polynomial fit captures the relationship between the response and explanatory variable much better than the linear fit:



Delete extra space between “Training” and “R^2”

```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('Training MSE linear: %.3f, quadratic: %.3f' % (
...         mean_squared_error(y, y_lin_pred),
...         mean_squared_error(y, y_quad_pred)))
Training MSE linear: 569.780, quadratic: 61.330
>>> print('Training  R^2 linear: %.3f, quadratic: %.3f' % (
...         r2_score(y, y_lin_pred),
...         r2_score(y, y_quad_pred)))
Training  R^2 linear: 0.832, quadratic: 0.982
```

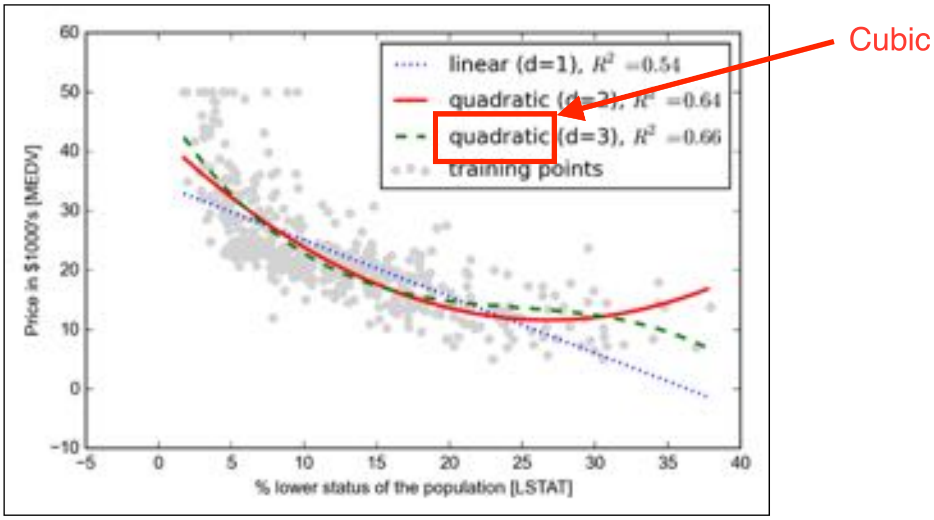
As we can see after executing the preceding code, the MSE decreased from 570 (linear fit) to 61 (quadratic fit), and the coefficient of determination reflects a closer fit to the quadratic model ($R^2 = 0.982$) as opposed to the linear fit ($R^2 = 0.832$) in this particular toy problem.

Modeling nonlinear relationships in the Housing Dataset

After we discussed how to construct polynomial features to fit nonlinear relationships in a toy problem, let's now take a look at a more concrete example and apply those concepts to the data in the *Housing Dataset*. By executing the following code, we will model the relationship between house prices and LSTAT (percent lower status of the population) using second degree (quadratic) and third degree (cubic) polynomials and compare it to a linear fit.

```
>>> plt.plot(X_fit, y_cubic_fit,
...          label='cubic (d=3), $R^2=%.2f$'
...            % cubic_r2,
...          color='green',
...          lw=2,
...          linestyle='--')
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

As we can see in the resulting plot, the cubic fit captures the relationship between the house prices and **LSTAT** better than the linear and quadratic fit. However, we should be aware that adding more and more polynomial features increases the complexity of a model and therefore increases the chance of overfitting. Thus, in practice, it is always recommended that you evaluate the performance of the model on a separate test dataset to estimate the generalization performance:



Cubic

In addition, polynomial features are not always the best choice for modeling nonlinear relationships. For example, just by looking at the **MEDV-LSTAT** scatterplot, we could propose that a log transformation of the **LSTAT** feature variable and the square root of **MEDV** may project the data onto a linear feature space suitable for a linear regression fit. Let's test this hypothesis by executing the following code:

```
# transform features
>>> X_log = np.log(X)
```

Dealing with nonlinear relationships using random forests

In this section, we are going to take a look at **random forest** regression, which is conceptually different from the previous regression models in this chapter. A random forest, which is an ensemble of multiple **decision trees**, can be understood as the sum of piecewise linear functions in contrast to the global linear and polynomial regression models that we discussed previously. In other words, via the decision tree algorithm, we are subdividing the input space into smaller regions that become more *manageable*.

Decision tree regression

An advantage of the decision tree algorithm is that it does not require any transformation of the features if we are dealing with nonlinear data. We remember from *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, that we grow a decision tree by iteratively splitting its nodes until the leaves are pure or a stopping criterion is satisfied. When we used decision trees for classification, we defined entropy as a measure of impurity to determine which feature split maximizes the **Information Gain** (**IG**), which can be defined as follows for a binary split:

Comment by a reader: “In the Kindle edition,
the first D is weirdly angled backwards.
In the print edition, it looks
offset. I'm not sure what you are
referring to here.”

$$IG\left(D_{p,}x\right) = I\left(D_p\right) - \frac{1}{N_p} I$$

Here, x is the feature to perform the split, N_p is the number of samples in the parent node, I is the impurity function, D_p is the subset of training samples in the parent node, and D and D are the subsets of training samples in the left and right child node after the split. Remember that our goal is to find the feature split that maximizes the information gain, or in other words, we want to find the feature split that reduces the impurities in the child nodes. In *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we used *entropy* as a measure of impurity, which is a useful criterion for classification. To use a decision tree for regression, we will replace entropy as the impurity measure of a node t by the MSE:

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t \right)^2$$

```
>>> print('MSE train: %.3f, test: %.3f' % (
...       mean_squared_error(y_train, y_train_pred),
...       mean_squared_error(y_test, y_test_pred)))
>>> print('R^2 train: %.3f, test: %.3f' % (
...       r2_score(y_train, y_train_pred),
...       r2_score(y_test, y_test_pred)))
1.642 MSE train: ~~3.235~~, test: 11.635
R^2 train: 0.960, test: 0.871
```

Unfortunately, we see that the random forest tends to overfit the training data. However, it's still able to explain the relationship between the target and explanatory variables relatively well ($R^2 = 0.871$ on the test dataset).

Lastly, let's also take a look at the residuals of the prediction:

```
>>> plt.scatter(y_train_pred,
...             y_train_pred - y_train,
...             c='black',
...             marker='o',
...             s=35,
...             alpha=0.5,
...             label='Training data')
>>> plt.scatter(y_test_pred,
...             y_test_pred - y_test,
...             c='lightgreen',
...             marker='s',
...             s=35,
...             alpha=0.7,
...             label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

Another problem with k-means is that one or more clusters can be empty. Note that this problem does not exist for k-medoids or fuzzy C-means, an algorithm that we will discuss in the next subsection. However, this problem is accounted for in the current k-means implementation in scikit-learn. If a cluster is empty, the algorithm will search for the sample that is farthest away from the centroid of the empty cluster. Then it will reassign the centroid to be this farthest point.

> When we are applying k-means to real-world data using a Euclidean distance metric, we want to make sure that the features are measured on the same scale and apply z-score standardization or min-max scaling if necessary.

After we predicted the cluster labels `y_km` and discussed the challenges of the k-means algorithm, let's now visualize the clusters that k-means identified in the dataset together with the cluster centroids. These are stored under the `centers_` attribute of the fitted `KMeans` object:

```
>>> plt.scatter(X[y_km==0,0],
...             X[y_km ==0,1],
...             s=50,
...             c='lightgreen',
...             marker='s',
...             label='cluster 1')
>>> plt.scatter(X[y_km ==1,0],
...             X[y_km ==1,1],
...             s=50,
...             c='orange',
...             marker='o',
...             label='cluster 2')
>>> plt.scatter(X[y_km ==2,0],
...             X[y_km ==2,1],
...             s=50,
...             c='lightblue',
...             marker='v',
...             label='cluster 3')
>>> plt.scatter(km.cluster_centers_[:,0],
...             km.cluster_centers_[:,1],
...             s=250,
...             marker='*',
...             c='red',
...             label='centroids')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()
```

remove whitespace before “==“

> Other commonly used algorithms for agglomerative hierarchical clustering include **average linkage** and **Ward's linkage**. In average linkage, we merge the cluster pairs based on the minimum average distances between all group members in the two clusters. In Ward's method, those two clusters that lead to the minimum increase of the total within-cluster SSE are merged.

In this section, we will focus on agglomerative clustering using the complete linkage approach. This is an iterative procedure that can be summarized by the following steps:

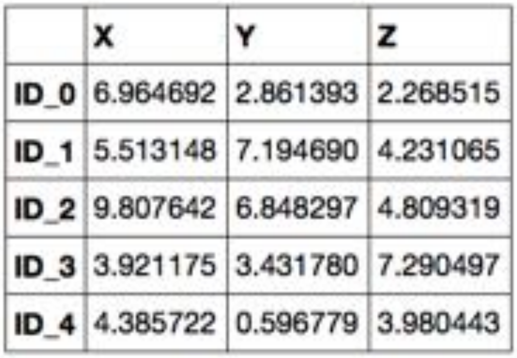
1. Compute the distance matrix of all samples.
2. Represent each data point as a singleton cluster.
3. Merge the two closest clusters based on the distance of the most dissimilar (distant) members.
4. “distance matrix” Update the ~~similarity matrix~~.
5. Repeat steps 2 to 4 until one single cluster remains.

Now we will discuss how to compute the distance matrix (step 1). But first, let's generate some random sample data to work with. The rows represent different observations (IDs 0 to 4), and the columns are the different features (X, Y, Z) of those samples:

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0','ID_1','ID_2','ID_3','ID_4']
>>> X = np.random.random_sample([5,3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

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

DataFrame containing the randomly generated samples:



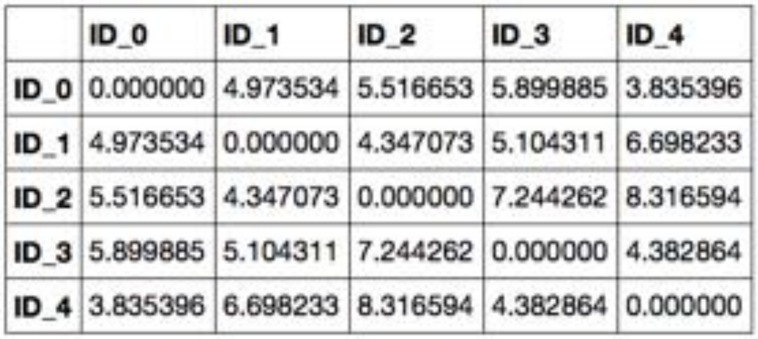
	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

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

Attaching dendrograms to a heat map

In practical applications, hierarchical clustering dendrograms are often used in combination with a **heat map**, which allows us to represent the individual values in the sample matrix with a color code. In this section, we will discuss how to attach a dendrogram to a heat map plot and order the rows in the heat map correspondingly.

However, attaching a dendrogram to a heat map can be a little bit tricky, so let's go through this procedure step by step:

1. We create a new `figure` object and define the *x* axis position, *y* axis position, width, and height of the dendrogram via the `add_axes` attribute. Furthermore, we rotate the dendrogram 90 degrees counter-clockwise. The code is as follows:

   Insert the following line in code formatting:

   ```
   >>> fig = plt.figure(~~figsize=(8,8))~~ figsize=(8,8), facecolor='white')
   >>> axd = fig.add_axes([0.09,0.1,0.2,0.6])
   >>> row_dendr = dendrogram(row_clusters, orientation='right')
   # note: for matplotlib >= v1.5.1, please use orientation='left'
   ```

2. Next we reorder the data in our initial `DataFrame` according to the clustering labels that can be accessed from the dendrogram object, which is essentially a Python dictionary, via the `leaves` key. The code is as follows:

   ```
   >>> df_rowclust = df.ix[row_dendr['leaves'][::-1]]
   ```

3. Now we construct the heat map from the reordered `DataFrame` and position it right next to the dendrogram:

   ```
   >>> axm = fig.add_axes([0.23,0.1,0.6,0.6])
   >>> cax = axm.matshow(df_rowclust,
   ...                 interpolation='nearest', cmap='hot_r')
   ```

4. Finally we will modify the aesthetics of the heat map by removing the axis ticks and hiding the axis spines. Also, we will add a color bar and assign the feature and sample names to the *x* and *y* axis tick labels, respectively. The code is as follows:

   ```
   >>> axd.set_xticks([])
   >>> axd.set_yticks([])
   >>> for i in axd.spines.values():
   ...     i.set_visible(False)
   >>> fig.colorbar(cax)
   >>> axm.set_xticklabels([''] + list(df_rowclust.columns))
   >>> axm.set_yticklabels([''] + list(df_rowclust.index))
   >>> plt.show()
   ```

Attaching dendrograms to a heat map

In practical applications, hierarchical clustering dendrograms are often used in combination with a **heat map**, which allows us to represent the individual values in the sample matrix with a color code. In this section, we will discuss how to attach a dendrogram to a heat map plot and order the rows in the heat map correspondingly.

However, attaching a dendrogram to a heat map can be a little bit tricky, so let's go through this procedure step by step:

1. We create a new `figure` object and define the *x* axis position, *y* axis position, width, and height of the dendrogram via the `add_axes` attribute. Furthermore, we rotate the dendrogram 90 degrees counter-clockwise. The code is as follows:

Insert the following line in code formatting:

```
>>> fig = plt.figure(~~figsize=(8,8))~~ figsize=(8,8), facecolor='white')
>>> axd = fig.add_axes([0.09,0.1,0.2,0.6])
>>> row_dendr = dendrogram(row_clusters, orientation='right')
# note: for matplotlib >= v1.5.1, please use orientation='left'
```

2. Next we reorder the data in our initial `DataFrame` according to the clustering labels that can be accessed from the dendrogram object, which is essentially a Python dictionary, via the `leaves` key. The code is as follows:

```
>>> df_rowclust = df.ix[row_dendr['leaves'][::-1]]
```

3. Now we construct the heat map from the reordered `DataFrame` and position it right next to the dendrogram:

```
>>> axm = fig.add_axes([0.23,0.1,0.6,0.6])
>>> cax = axm.matshow(df_rowclust,
...               interpolation='nearest', cmap='hot_r')
```

4. Finally we will modify the aesthetics of the heat map by removing the axis ticks and hiding the axis spines. Also, we will add a color bar and assign the feature and sample names to the *x* and *y* axis tick labels, respectively. The code is as follows:

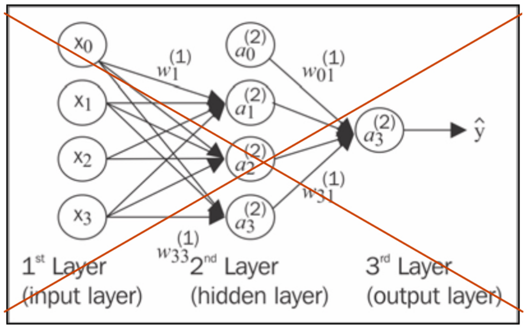
```
>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():
...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels([''] + list(df_rowclust.columns))
>>> axm.set_yticklabels([''] + list(df_rowclust.index))
>>> plt.show()
```

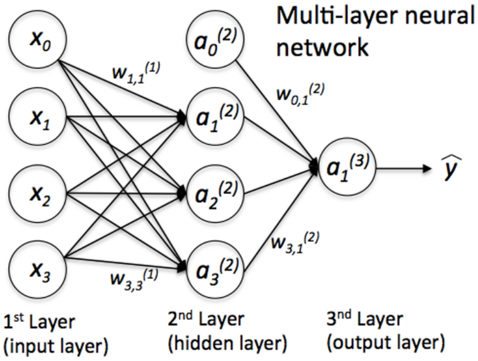
Note that although Adaline consists of two layers, one input layer and one output layer, it is called a single-layer network because of its single link between the input and output layers.

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 ha
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*.

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~~ layer is connected to all units in the input layers, we first calculate the activation $a_1^{(2)}$ as follows:

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

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

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

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

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

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:

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

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.

The way we read in the image might seem a little bit strange at first:

```
magic, n = struct.unpack('>II', lbpath.read(8))
labels = np.fromfile(lbpath, dtype=np.int8)
```

To understand how these two lines of code work, let's take a look at the dataset description from the MNIST website:

[offset]	*[type]*	*[value]*	*[description]*
0000	*32 bit integer*	*0x00000801(2049)*	*magic number (MSB first)*
0004	*32 bit integer*	*60000*	*number of items*
0008	*unsigned byte*	*??*	*label*
0009	*unsigned byte*	*??*	*label*
....			
xxxx	*unsigned byte*	*??*	*label*

Using the two lines of the preceding code, we first read in the *magic number*, which is a description of the file protocol as well as the *number of items* (*n*) from the file buffer before we read the following bytes into a NumPy array using the `fromfile` method. The `fmt` parameter value `>II` that we passed as an argument to `struct.unpack` has two parts:

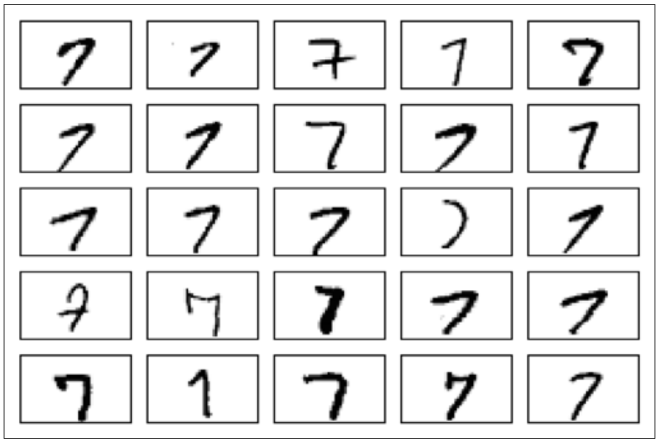
delete “the”

- `>`: This is ~~the~~ big-endian (defines the order in which a sequence of bytes is stored); if you are unfamiliar with the terms *big-endian* and *small-endian*, you can find an excellent article about *Endianness* on Wikipedia (`https://en.wikipedia.org/wiki/Endianness`).
- `I`: This is an unsigned integer.

By executing the following code, we will now load the 60,000 training instances as well as the 10,000 test samples from the `mnist` directory where we unzipped the MNIST dataset:

```
>>> X_train, y_train = load_mnist('mnist', kind='train')
>>> print('Rows: %d, columns: %d'
...        % (X_train.shape[0], X_train.shape[1]))
Rows: 60000, columns: 784

>>> X_test, y_test = load_mnist('mnist', kind='t10k')
>>> print('Rows: %d, columns: %d'
...        % (X_test.shape[0], X_test.shape[1]))
Rows: 10000, columns: 784
```

After executing the code, we should now see the first 25 variants of the digit 7.



Optionally, we can save the MNIST image data and labels as CSV files to open them in programs that do not support their special byte format. However, we should be aware that the CSV file format will take up substantially more space on your local drive, as listed here:

- `train_img.csv`: 109.5 MB
- `train_labels.csv`: 120 KB
- `test_img.csv`: 18.3 MB
- test_labels.csv ~~`test_labels`~~: 20 KB

If we decide to save those CSV files, we can execute the following code in our Python session after loading the MNIST data into NumPy arrays:

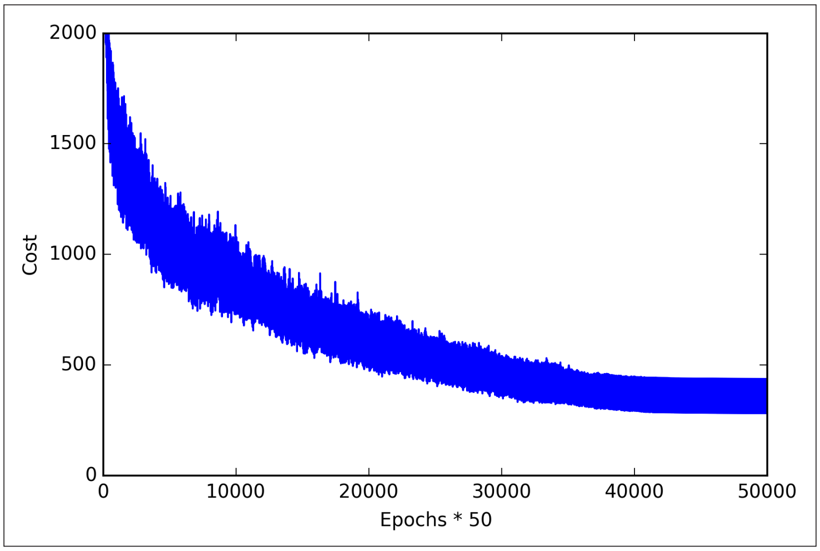
```
>>> np.savetxt('train_img.csv', X_train,
...            fmt='%i', delimiter=',')
>>> np.savetxt('train_labels.csv', y_train,
...            fmt='%i', delimiter=',')
>>> np.savetxt('test_img.csv', X_test,
...            fmt='%i', delimiter=',')
>>> np.savetxt('test_labels.csv', y_test,
...            fmt='%i', delimiter=',')
```

```
        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:                  X_data, y_enc = X_data[idx],  y_enc[:,idx
                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)    X_data[idx]
                cost = self._get_cost(y_enc=y_enc[:, idx],
                                      output=a3,
                                      w1=self.w1,
                                      w2=self.w2)
                self.cost_.append(cost)
```

```
>>> 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.

As we see in the following plot, the graph of the cost function looks very noisy. This is due to the fact that we trained our neural network with mini-batch learning, a variant of stochastic gradient descent.



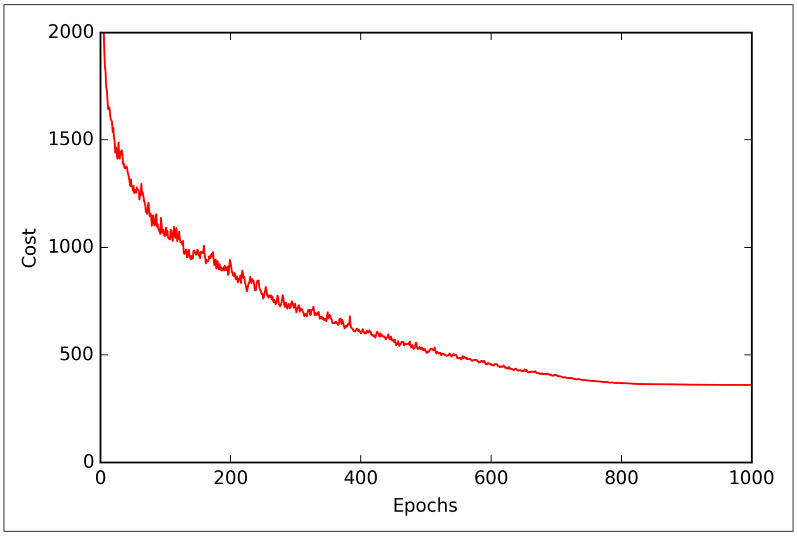
lease see the IPython otebook for an updated igure

Although we can already see in the plot that the optimization algorithm converged after approximately 800 epochs (*40,000/50 = 800*), let's plot a smoother version of the cost function against the number of epochs by averaging over the mini-batch intervals. The code is as follows:

```
>>> batches = np.array_split(range(len(nn.cost_)), 1000)
>>> cost_ary = np.array(nn.cost_)
>>> cost_avgs = [np.mean(cost_ary[i]) for i in batches]

>>> plt.plot(range(len(cost_avgs)),
...          cost_avgs,
...          color='red')
>>> plt.ylim([0, 2000])
>>> plt.ylabel('Cost')
>>> plt.xlabel('Epochs')
>>> plt.tight_layout()
>>> plt.show()
```

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

Please see the IPython notebook for an updated figure



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

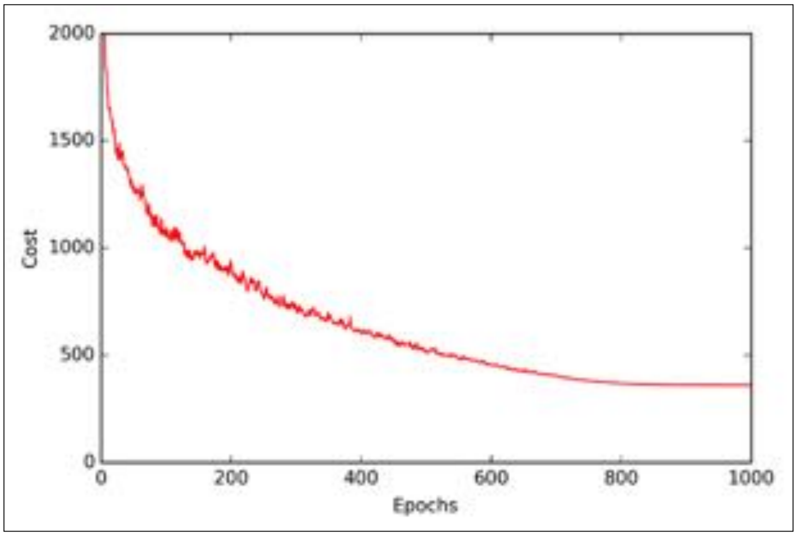
```
>>> y_train_pred = nn.predict(X_train)
>>> acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (acc * 100))
Training accuracy: ~~97.74%~~
                   97.59%
```

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

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
>>> print('Test accuracy: %.2f%%' % (acc * 100))
Test accuracy: ~~96.18%~~
               95.62%
```

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

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]
Test >>> print('Training accuracy: %.2f%%' % (acc * 100))
     Test accuracy: 96.18%
```

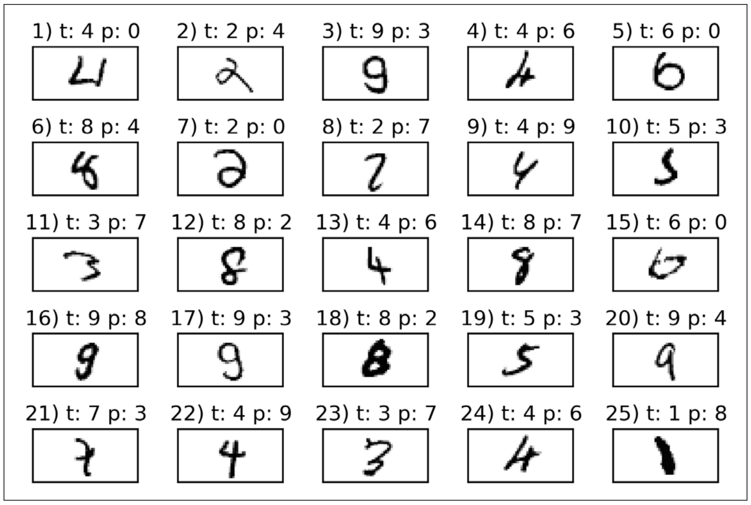
Based on the small discrepancy between training and test accuracy, we can conclude that the model only slightly overfits the training data. To further fine-tune the model, we could change the number of hidden units, values of the regularization parameters, learning rate, values of the decrease constant,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

Now, let's take a look at some of the images that our MLP struggles with:

```
>>> miscl_img = X_test[y_test != y_test_pred][:25]
>>> correct_lab = y_test[y_test != y_test_pred][:25]
>>> miscl_lab= y_test_pred[y_test != y_test_pred][:25]

>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = miscl_img[i].reshape(28, 28)
...     ax[i].imshow(img,
...                  cmap='Greys',
...                  interpolation='nearest')
...     ax[i].set_title('%d) t: %d p: %d'
...                     % (i+1, correct_lab[i], miscl_lab[i]))
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

We should now see a 5×5 subplot matrix where the first number in the subtitles indicates the plot index, the second number indicates the true class label (t), and the third number stands for the predicted class label (p).

Please see the IPython notebook for an updated figure



As we can see in the preceding figure, some of those images are even challenging for us humans to classify correctly. For example, we can see that the digit ~~9~~ is 0 classified as a 6 ~~3 or 8 if the lower part of the digit has a hook-like curvature (subplots 3, 16, and 17).~~ “in subplot 15.”

Training an artificial neural network

Now that we have seen a neural network in action and have gained a basic understanding of how it works by looking over the code, let's dig a little bit deeper into some of the concepts, such as the logistic cost function and the backpropagation algorithm that we implemented to learn the weights.

Computing the logistic cost function

The logistic cost function that we implemented as the `_get_cost` method is actually pretty simple to follow since it is the same cost function that we described in the logistic regression section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*.

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)$$

Here, $a^{(i)}$ is the sigmoid activation of the i^{th} unit in one of the layers which we compute in the forward propagation step:

$$a^{(i)} = \phi\left(z^{(i)}\right)$$

Now, let's add a **regularization** term, which allows us to reduce the degree of overfitting. As you will recall from earlier chapters, the L2 and L1 regularization terms are defined as follows (remember that we don't regularize the bias units):

$$L2 = \lambda \|\boldsymbol{w}\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2 \text{ and } L1 = \lambda \|\boldsymbol{w}\|_1^1 = \lambda \sum_{j=1}^{m} \left|w_j\right|$$

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

There should be a minus sign in front of the opening bracket

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

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:

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

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:

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

expression The following ~~equation~~ represents the L2-penalty term:

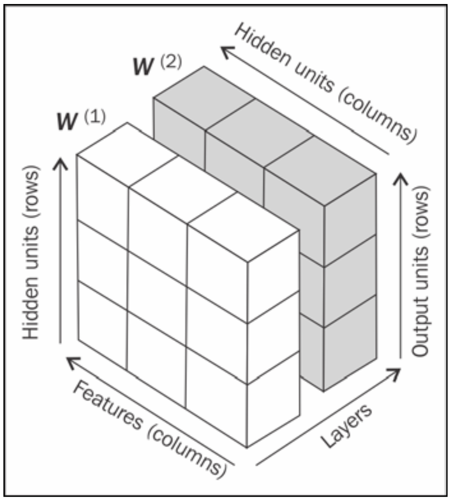
$$\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$$

Remember that our goal is to minimize the cost function $J(\boldsymbol{w})$. Thus, we need to calculate the partial derivative of matrix $\boldsymbol{W}$ with respect to each weight for every layer in the network:

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(\boldsymbol{W})$$

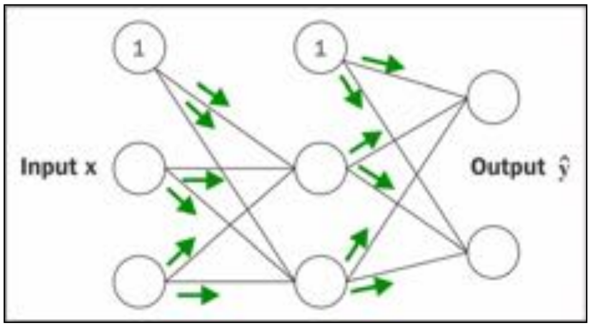
In the next section, we will talk about the backpropagation algorithm, which allows us to calculate these partial derivatives to minimize the cost function.

Note that $\boldsymbol{W}$ consists of multiple matrices. In a multi-layer perceptron with one layer hidden ~~unit~~, we have the weight matrix $\boldsymbol{W}^{(1)}$, which connects the input to the hidden layer, and $\boldsymbol{W}^{(2)}$, which connects the hidden layer to the output layer. An intuitive visualization of the matrix $\boldsymbol{W}$ is provided in the following figure:



In this simplified figure, it may seem that both $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$ have the same number of rows and columns, which is typically not the case unless we initialize an MLP with the same number of hidden units, output units, and input features.

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



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

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

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

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

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

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

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

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

Although, it is not important to follow the next equations, you may be curious as to how I obtained the derivative of the activation function. I summarized the derivation step by step here:

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

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

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

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

$$= \phi(z) - \left(\phi(z)\right)^2$$

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

$$= a(1-a)$$

To better understand how we compute the $\delta^{(2)}$ term, let's walk through it in more detail. In the preceding equation, we multiplied the transpose $\left(\boldsymbol{W}^{(2)}\right)^T$ of the $t \times h$ dimensional matrix $\boldsymbol{W}^{(2)}$; t is the number of output class labels and h is the number of hidden units~~)~~. Now, $\left(\boldsymbol{W}^{(2)}\right)^T$ becomes an $h \times t$ dimensional matrix with $\delta^{(2)}$, which is a $t \times 1$ dimensional vector. We then performed a pair-wise multiplication between $\left(\boldsymbol{W}^{(2)}\right)^T \delta^{(3)}$ and $\left(a^{(2)} * \left(1-a^{(2)}\right)\right)$, which is also a $t \times 1$ dimensional vector. Eventually, after obtaining the δ terms, we can now write the derivation of the cost function as follows:

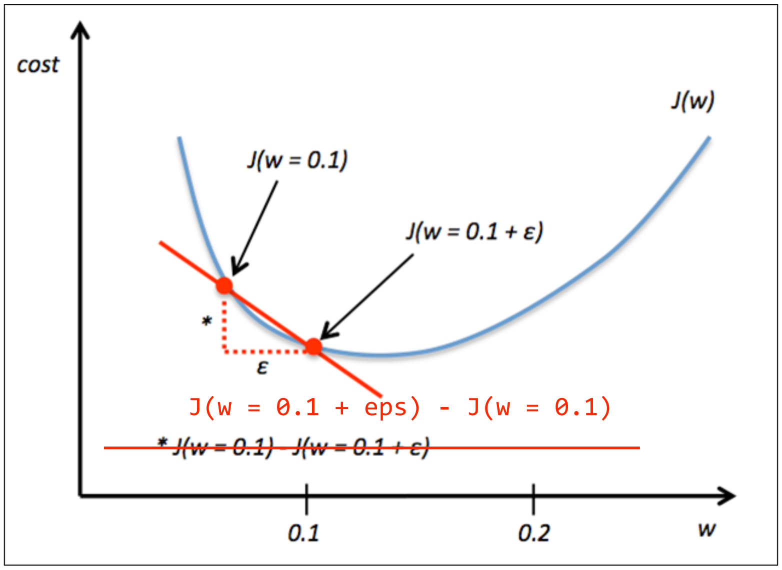
delete “)”

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\boldsymbol{W}) = a_j^{(l)} \delta_i^{(l+1)}$$

Remember that we are updating the weights by taking an opposite step towards the direction of the gradient. In gradient checking, we compare this analytical solution to a numerically approximated gradient:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\boldsymbol{W}) \approx \frac{J\left(w_{i,j}^{(l)} + \varepsilon\right) - J\left(w_{i,j}^{(l)}\right)}{\varepsilon}$$

Here, ε is typically a very small number, for example 1e-5 (note that 1e-5 is just a more convenient notation for 0.00001). Intuitively, we can think of this finite difference approximation as the slope of the secant line connecting the points of the cost function for the two weights w and $w + \varepsilon$ (both are scalar values), as shown in the following figure. We are omitting the superscripts and subscripts for simplicity.



An even better approach that yields a more accurate approximation of the gradient is to compute the symmetric (or centered) difference quotient given by the two-point formula:

$$\frac{J\left(w_{i,j}^{(l)} + \varepsilon\right) - J\left(w_{i,j}^{(l)} - \varepsilon\right)}{2\varepsilon}$$

Working with array structures

In this section, we will discuss how to use array structures in Theano using its `tensor` module. By executing the following code, we will create a simple 2 x 3 matrix, and calculate the column sums using Theano's optimized tensor expressions:

```
>>> import numpy as np

# initialize
>>> x = T.fmatrix(name='x')
>>> x_sum = T.sum(x, axis=0)

# compile
>>> calc_sum = theano.function(inputs=[x], outputs=x_sum)

# execute (Python list)
>>> ary = [[1, 2, 3], [1, 2, 3]]
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2.  4.  6.]

# execute (NumPy array)
>>> ary = np.array([[1, 2, 3], [1, 2, 3]],
...                dtype=theano.config.floatX)
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2.  4.  6.]
```

```
Add the following lines:

# if you are running Theano on 64 bit mode,
# you need to use dmatrix instead of fmatrix
```

As we saw earlier, there are just three basic steps that we have to follow when we are using Theano: defining the variable, compiling the code, and executing it. The preceding example shows that Theano can work with both Python and NumPy types: `list` and `numpy.ndarray`.

> Note that we used the optional name argument (here, `x`) when we created the `fmatrix` TensorVariable, which can be helpful to debug our code or print the Theano graph. For example, if we'd print the `fmatrix` symbol `x` without giving it a `name`, the `print` function would return its `TensorType`:
>
> ```
> >>> print(x)
> <TensorType(float32, matrix)>
> ```
>
> However, if the `TensorVariable` was initialized with a `name` argument `x` as in our preceding example, it would be returned by the `print` function:
>
> ```
> >>> print(x)
> x
> ```
>
> The `TensorType` can be accessed via the `type` method:
>
> ```
> >>> print(x.type())
> <TensorType(float32, matrix)>
> ```

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:

$$P(y=i\,|\,z)=\phi_{softmax}(z)=\frac{e_i^z}{\sum_{m=1}^{M}e_m^z}$$

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

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

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

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.

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

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

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

The advantage of the hyperbolic tangent over the logistic function is that it has a broader output spectrum and ranges the open interval (-1, 1), which can improve the convergence of the back propagation algorithm (C. M. Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995, pp. 500-501). In contrast, the logistic function returns an output signal that ranges the open interval (0, 1). For an intuitive comparison of the logistic function and the hyperbolic tangent, let's plot two sigmoid functions ~~in a one-dimensional space:~~

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

>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)

>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)

>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('net input $z$')
>>> plt.ylabel('activation $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle='--')
>>> plt.axhline(0.5, color='black', linestyle='--')
>>> plt.axhline(0, color='black', linestyle='--')
>>> plt.axhline(-1, color='black', linestyle='--')

>>> plt.plot(z, tanh_act,
...          linewidth=2,
...          color='black',
...          label='tanh')
>>> plt.plot(z, log_act,
...          linewidth=2,
...          color='lightgreen',
...          label='logistic')

>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```