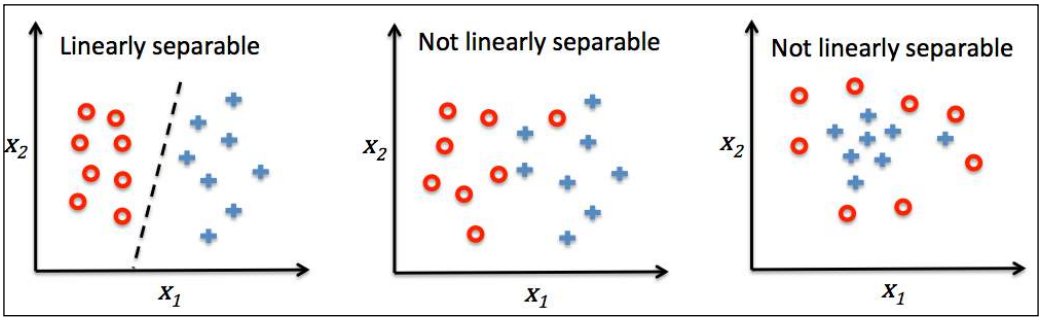
Let's assume that $x_j^{(i)} = 0.5$, and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1 so that the activation $x_j^{(i)} \times w_j^{(i)}$ will be more positive the next time we encounter this sample and thus will be more likely to be above the threshold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

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

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

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

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



Downloading the example code

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

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:

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

Here, λ is the so-called regularization parameter.

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

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

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

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

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

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

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

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

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  1.0  2.0  3.0  4.0
1  5.0  6.0  NaN  8.0
2 10.0 11.0 12.0  NaN
```

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

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

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

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

Although scikit-learn was developed for working with NumPy arrays, it can sometimes be more convenient to preprocess data using pandas' `DataFrame`. We can always access the underlying NumPy array of the `DataFrame` via the `values` attribute before we feed it into a scikit-learn estimator:

```
>>> df.values
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  nan,   8.],
       [ 10.,  11.,  12.,  nan]])
```

Eliminating samples or features with missing values

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

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

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

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

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

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

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

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

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

Imputing missing values

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

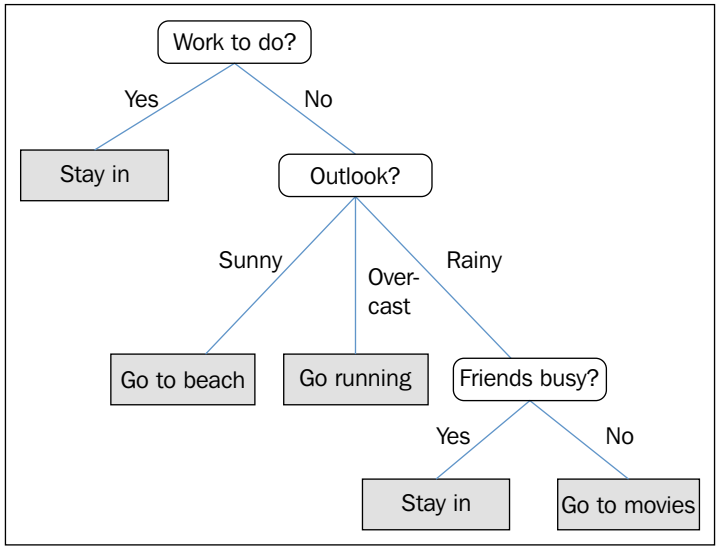
```
>>> from sklearn.preprocessing import Imputer
>>> imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imr = imr.fit(df)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,7.5,   8.],
       [ 10.,  11.,  12.,  6 ]])
```

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

Understanding the scikit-learn estimator API

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

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



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

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

Assessing feature importance with random forests

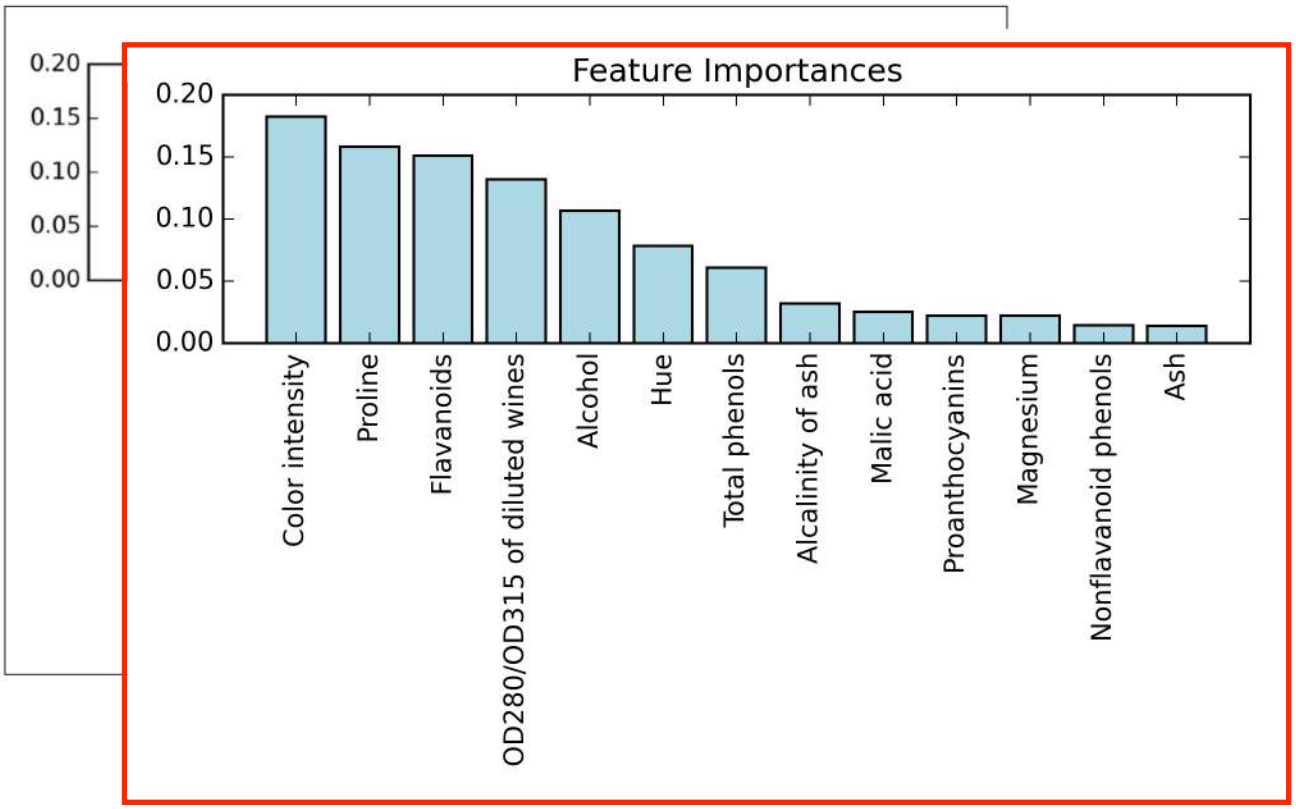
In the previous sections, you learned how to use L1 regularization to zero out irrelevant features via logistic regression and use the SBS algorithm for feature selection. Another useful approach to select relevant features from a dataset is to use a random forest, an ensemble technique that we introduced 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                        0.182503
2) Malic acid                     0.158574
3) Ash                            0.150954
4) Alcalinity of ash              0.131983
5) Magnesium                      0.106564
6) Total phenols                  0.078249
7) Flavanoids                     0.060717
8) Nonflavanoid phenols           0.032039
9) Proanthocyanins                0.025385
10) Color intensity               0.022369
11) Hue                           0.022070
```

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

```
12) OD280/OD315 of diluted wines  0.014655
13) Proline                       0.013933
>>> plt.title('Feature Importances')
>>> plt.bar(range(X_train.shape[1]),
...         importances[indices],
...         color='lightblue',
...         align='center')
>>> plt.xticks(range(X_train.shape[1]),
...            feat_labels, rotation=90)      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 alcohol content of wine is the most discriminative feature in the dataset based on the average impurity decrease in the 10,000 decision trees. Interestingly, the three top-ranked features in the preceding plot are also among the top five features in the selection by the SBS algorithm that we implemented in the previous section. However, as far as interpretability is concerned, the random forest technique comes with an important *gotcha* that is worth mentioning. For instance, if two or more features are highly correlated, one feature may be ranked very highly while the information of the other feature(s) may not be fully captured. On the other hand, we don't need to be concerned about this problem if we are merely interested in the predictive performance of a model rather than the interpretation of feature importances. To conclude this section about feature importances and random forests, it is worth mentioning that scikit-learn also implements a `transform` method that selects features based on a user-specified threshold after model fitting, which is useful if we want to use the `RandomForestClassifier` as a feature selector and intermediate step in a scikit-learn pipeline, which allows us to connect different preprocessing steps with an estimator, as we will see in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*. For example, we could set the threshold to 0.15 to reduce the dataset to the 3 most important features, **Alcohol**, **Malic acid**, and **Ash** using the following code:

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

Summary

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

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

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

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

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

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

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

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

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

for an example why this can be a problem.

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

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

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

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

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

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

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

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

Serializing fitted scikit-learn estimators

Training a machine learning model can be computationally quite expensive, as we have seen in *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*. Surely, we don't want to train our model every time we close our Python interpreter and want to make a new prediction or reload our web application? One option for **model persistence** is Python's in-built pickle module (`https://docs.python.org/3.4/library/pickle.html`), which allows us to serialize and de-serialize Python object structures to compact byte code, so that we can save our classifier in its current state and reload it if we want to classify new samples without needing to learn the model from the training data all over again. Before you execute the following code, please make sure that you have trained the out-of-core logistic regression model from the last section of *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*, and have it ready in your current Python session:

```
>>> import pickle
>>> import os
>>> dest = os.path.join('movieclassifier', 'pkl_objects')
>>> if not os.path.exists(dest):
...     os.makedirs(dest)
>>> pickle.dump(stop,
...           open(os.path.join(dest, 'stopwords.pkl'),'wb'),
...           protocol=4)
>>> pickle.dump(clf,
...           open(os.path.join(dest, 'classifier.pkl'), 'wb'),
...           protocol=4)
```

Using the preceding code, we created a `movieclassifier` directory where we will later store the files and data for our web application. Within this `movieclassifier` directory, we created a `pkl_objects` subdirectory to save the serialized Python objects to our local drive. Via pickle's `dump` method, we then serialized the trained logistic regression model as well as the stop word set from the NLTK library so that we don't have to install the NLTK vocabulary on our server. The `dump` method takes as its first argument the object that we want to pickle, and for the second argument we provided an open file object that the Python object will be written to. Via the `wb` argument inside the `open` function, we opened the file in binary mode for pickle, and we set `protocol=4` to choose the latest and most efficient pickle protocol that has been added to Python 3.4. (If you have problems using protocol 4, please check if you are using the latest Python 3 version install. Alternatively, you may consider choosing a lower protocol number.)

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

DataFrame containing the randomly generated samples:

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

Performing hierarchical clustering on a distance matrix

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

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

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

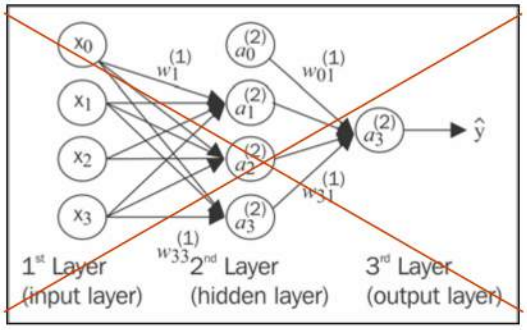
	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

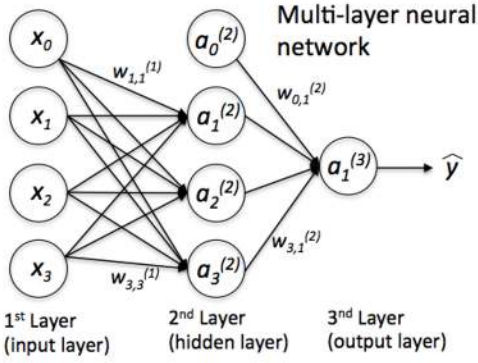
Note that although Adaline consists of two layers, one input layer and one output layer, it is called a single-layer network because of its single link between the input and output layers.

Introducing the multi-layer neural network architecture

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

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

            return grad1, grad2

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

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

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

            for i in range(self.epochs):

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

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

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

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

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

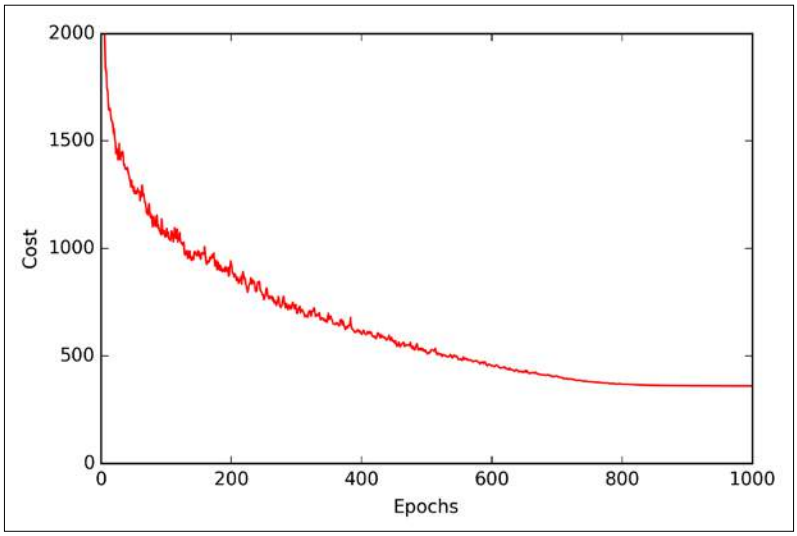
X_data, y_enc = X_data[idx], y_enc[:,idx]

X_data[idx]


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

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

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



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

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

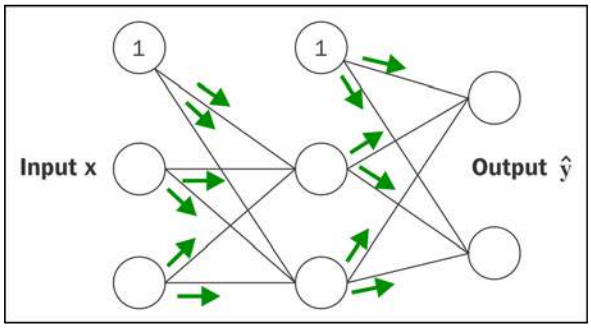
As we can see, the model classifies most of the training digits correctly, but how does it generalize to data that it has not seen before? Let's calculate the accuracy on 10,000 images in the test dataset:

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

Test

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Choosing activation functions for feedforward neural networks

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

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

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

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

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