Some people asked me: “So, you have been coding
in Python for just a year?” That’s really not true, I’ve been programming in Python
since 2011 or so :P

Please replace “He has a yearlong experience in Python programming”
by “He has many years of experience with coding in Python”

About the Author

Sebastian Raschka is a PhD student at Michigan State University, who develops new computational methods in the field of computational biology. He has been ranked as the number one most influential data scientist on GitHub by Analytics Vidhya. ~~He has a yearlong experience in Python programming~~ and he has conducted several seminars on the practical applications of data science and machine learning. Talking and writing about data science, machine learning, and Python really motivated Sebastian to write this book in order to help people develop data-driven solutions without necessarily needing to have a machine learning background.

He has also actively contributed to open source projects and methods that he implemented, which are now successfully used in machine learning competitions, such as Kaggle. In his free time, he works on models for sports predictions, and if he is not in front of the computer, he enjoys playing sports.

I would like to thank my professors, Arun Ross and Pang-Ning Tan, and many others who inspired me and kindled my great interest in pattern classification, machine learning, and data mining.

I would like to take this opportunity to thank the great Python community and developers of open source packages who helped me create the perfect environment for scientific research and data science.

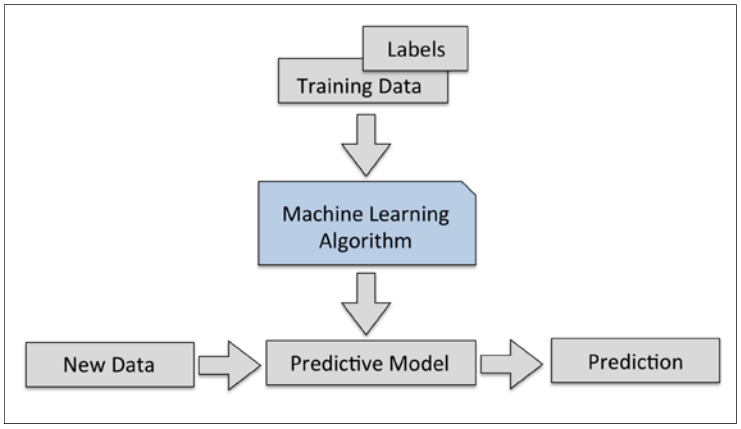
A special thanks goes to the core developers of scikit-learn. As a contributor to this project, I had the pleasure to work with great people, who are not only very knowledgeable when it comes to machine learning, but are also excellent programmers.

Lastly, I want to thank you all for showing an interest in this book, and I sincerely hope that I can pass on my enthusiasm to join the great Python and machine learning communities.

Making predictions about the future with supervised learning

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

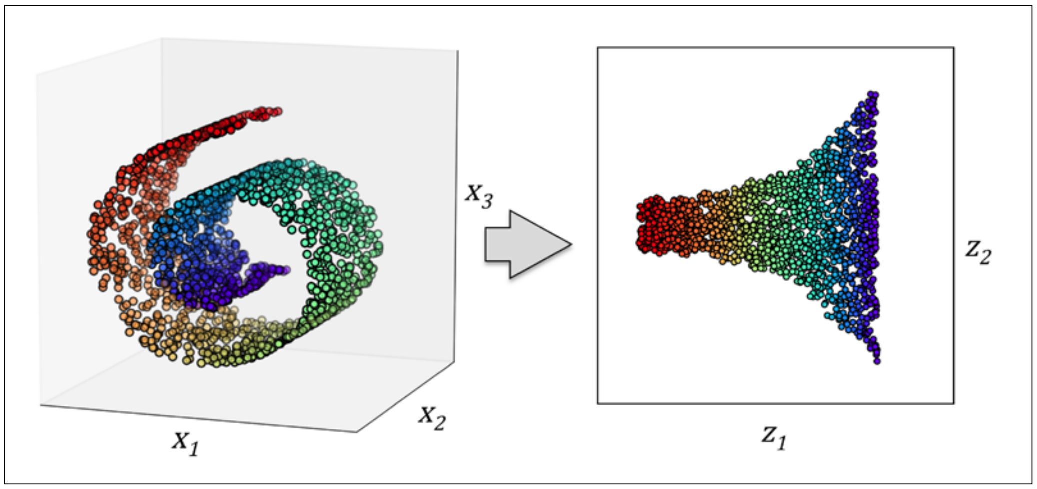
Considering the example of e-mail spam filtering, we can train a model using a supervised machine learning algorithm on a corpus of labeled e-mail, e-mail that are correctly marked as spam or not-spam, to predict whether a new e-mail belongs to either of the two categories. A supervised learning task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



Classification for predicting class labels

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

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



An introduction to the basic terminology and notations

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

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

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

The last two chapters

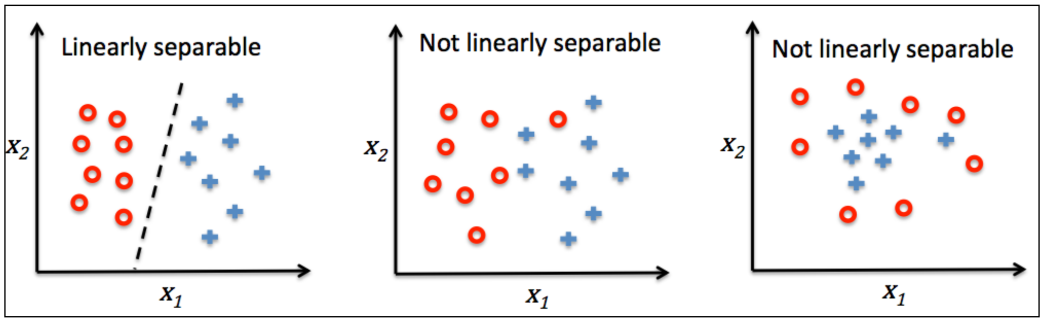
Let's assume that $x_j^{(i)} = 0.5$, and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1 so that the activation $x_j^{(i)} = w_j^{(i)}$ will be more positive the next time we encounter this sample and thus will be more likely to be above the threshold of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



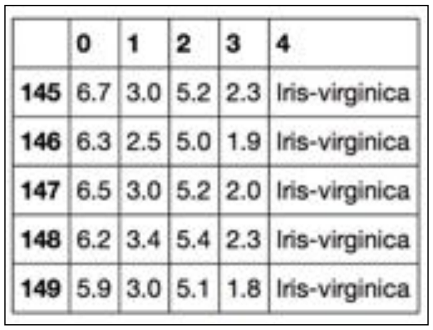
Downloading the example code

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.

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

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

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



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

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

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


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

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

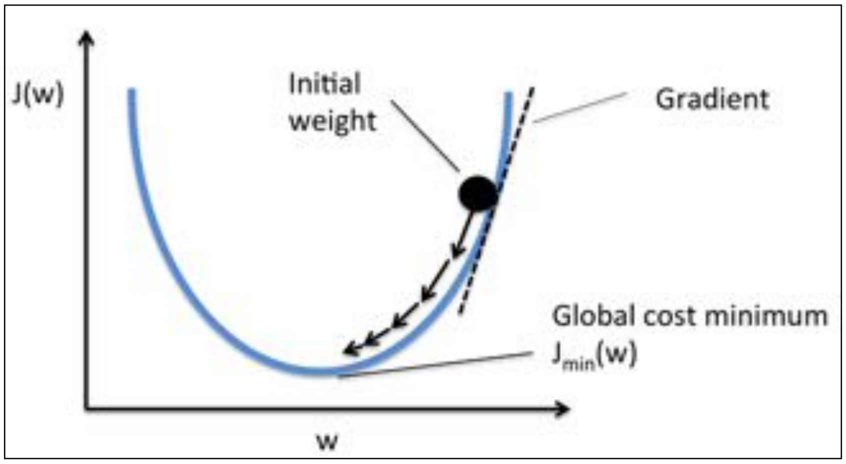
Minimizing cost functions with gradient descent

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

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

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

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



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

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

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

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

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

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

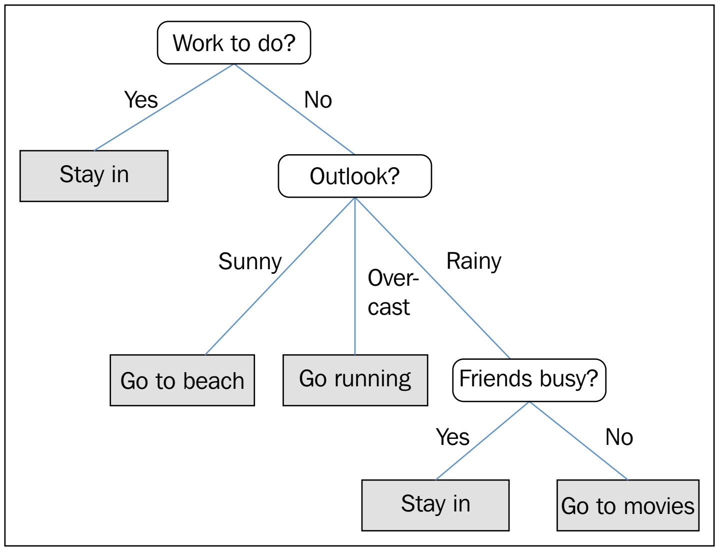
Although stochastic gradient descent can be considered as an approximation of gradient descent, it typically reaches convergence much faster because of the more frequent weight updates. Since each gradient is calculated based on a single training example, the error surface is noisier than in gradient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

In stochastic gradient descent implementations, the fixed learning rate η is often replaced by an adaptive learning rate that decreases over time, for example, $\frac{c_1}{[number\ of\ iterations] + c_2}$ where c_1 and c_2 are constants. Note that stochastic gradient descent does not reach the global minimum but an area very close to it. By using an adaptive learning rate, we can achieve further annealing to a better global minimum

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

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

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.

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

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

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

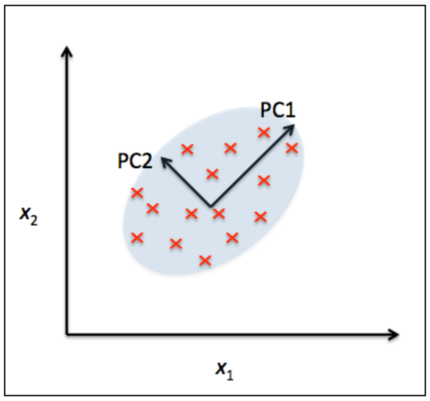
```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                            metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))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

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

Unsupervised dimensionality reduction via principal component analysis

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

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



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()
```

Summary

In this chapter, we looked at some of the most popular and widely used techniques for ensemble learning. Ensemble methods combine different classification models to cancel out their individual ~~weakness,~~ which often results in stable and well-performing models that are very attractive for industrial applications as well as machine learning competitions. weaknesses

In the beginning of this chapter, we implemented a `MajorityVoteClassifier` in Python that allows us to combine different ~~algorithm~~ for classification. We then looked at bagging, a useful technique to reduce the variance of a model by drawing random bootstrap samples from the training set and combining the individually trained classifiers via majority vote. Then we discussed AdaBoost, which is an algorithm that is based on weak learners that subsequently learn from mistakes. algorithms

Throughout the previous chapters, we discussed different learning algorithms, tuning, and evaluation techniques. In the following chapter, we will look at a particular application of machine learning, sentiment analysis, which has certainly become an interesting topic in the era of the Internet and social media.

The Porter stemming algorithm is probably the oldest and simplest stemming algorithm. Other popular stemming algorithms include the newer **Snowball stemmer** (Porter2 or "English" stemmer) or the **Lancaster stemmer** (Paice-Husk stemmer), which is faster but also more aggressive than the Porter stemmer. Those alternative stemming algorithms are also available through the NLTK package (`http://www.nltk.org/api/nltk.stem.html`).

While stemming can create non-real words, such as `thu`, (from `thus`) as shown in the previous example, a technique called **lemmatization** aims to obtain the canonical (grammatically correct) forms of individual words—the so-called **lemmas**. However, lemmatization is computationally more difficult and expensive compared to stemming and, in practice, it has been observed that stemming and lemmatization have little impact on the performance of text classification (Michal Toman, Roman Tesar, and Karel Jezek. *Influence of word normalization on text classification*. Proceedings of InSciT, pages 354–358, 2006).

Before we jump into the next section where we will train a machine learning model using the bag-of-words model, let us briefly talk about another useful topic called **stop-word removal**. Stop-words are simply those words that are extremely common in all sorts of texts and likely bear no (or only little) useful information that can be used to distinguish between different classes of documents. Examples of stop-words are *is*, *and*, *has*, and the like. Removing stop-words can be useful if we are working with raw or normalized term frequencies rather than tf-idfs, which are already downweighting frequently occurring words.

In order to remove stop-words from the movie reviews, we will use the set of 127 English stop-words that is available from the NLTK library, which can be obtained by calling the `nltk.download` function:

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

After we have downloaded the stop-words set, we can load and apply the English stop-word set as follows:

```
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>>  [w for w in tokenizer_porter('a runner likes running and runs a
lot')[-10:] if w not in stop]

['runner', 'like', 'run', 'run', 'lot']
```

As we can see, the accuracy of the model is 87 percent, slightly below the accuracy that we achieved in the previous section using the grid search for hyperparameter tuning. However, out-of-core learning is very memory-efficient and took less than a minute to complete. Finally, we can use the last 5,000 documents to update our model:

```
>>> clf = clf.partial_fit(X_test, y_test)
```

If you are planning to continue directly with *Chapter 9*, *Embedding a Machine Learning Model into a Web Application*, I recommend you to keep the current Python session
we will use open. In the next chapter, ~~will use~~ the model that we just trained to learn how to save it to disk for later use and embed it into a web application.

Although the bag-of-words model is still the most commonly used model for text classification, it does not consider sentence structure and grammar. A popular extension of the bag-of-words model is **Latent Dirichlet allocation**, which is a topic model that considers the latent semantics of words (D. M. Blei, A. Y. Ng, and M. I. Jordan. *Latent Dirichlet allocation*. The Journal of machine Learning research, 3:993–1022, 2003).

A more modern alternative to the bag-of-words model is **word2vec**, an algorithm that Google released in 2013 (T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. arXiv preprint arXiv:1301.3781, 2013). The word2vec algorithm is an unsupervised learning algorithm based on neural networks that attempts to automatically learn the relationship between words. The idea behind word2vec is to put words that have similar meanings into similar clusters; via clever vector-spacing, the model can reproduce certain words using simple vector math, for example, *king* – *man* + *woman* = *queen*.

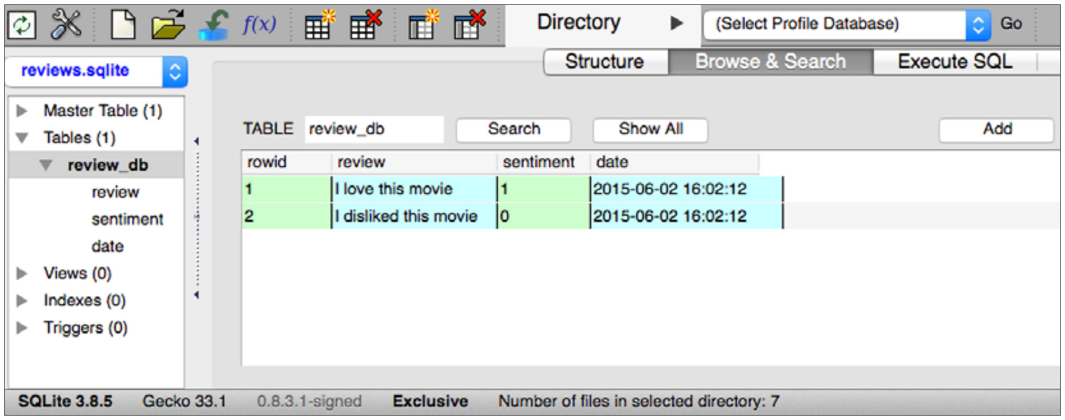
The original C-implementation, with useful links to the relevant papers and alternative implementations, can be found at `https://code.google.com/p/word2vec/`.

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~~)~~ .)



rowid	review	sentiment	date
1	I love this movie	1	2015-06-02 16:02:12
2	I disliked this movie	0	2015-06-02 16:02:12

SQLite 3.8.5 Gecko 33.1 0.8.3.1-signed Exclusive Number of files in selected directory: 7

Developing a web application with Flask

After we have prepared the code to classify movie reviews in the previous subsection, let's discuss the basics of the Flask web framework to develop our web application. After Armin Ronacher's initial release of Flask in 2010, the framework has gained huge popularity over the years and examples of popular applications that make use of Flask include LinkedIn and Pinterest. Since Flask is written in Python, it provides us Python programmers with a convenient interface for embedding existing Python code such as our movie classifier.

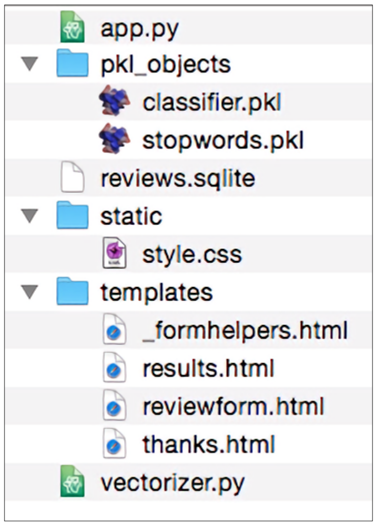
nown as a
icroframework

Flask is also known as a *microframework*, which means that its core is kept lean and simple but can be easily extended with other libraries. Although the learning curve of the lightweight Flask API is not nearly as steep as those of other popular Python web frameworks, such as Django, I encourage you to take a look at the official Flask documentation at `http://flask.pocoo.org/docs/0.10/` to learn more about its functionality.

If the Flask library is not already installed in your current Python environment, you can simply install it via pip from your terminal (at the time of writing, the latest stable release was Version 0.10.1):

```
pip install flask
```

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
```

Now, it would be a good idea to start the web app locally from our terminal via the following command before we advance to the next subsection and deploy it on a public web server:

```
python3 app.py
```

After we have finished testing our app, we also shouldn't forget to remove the `debug=True` argument in the `app.run()` command of our `app.py` script.

Deploying the web application to a public server

After we have tested the web application locally, we are now ready to deploy our web application onto a public web server. For this tutorial, we will be using the **PythonAnywhere** web hosting service, which specializes in the hosting of Python web applications and makes it extremely simple and hassle-free. Furthermore, PythonAnywhere offers a beginner account option that lets us run a single web application free of charge.

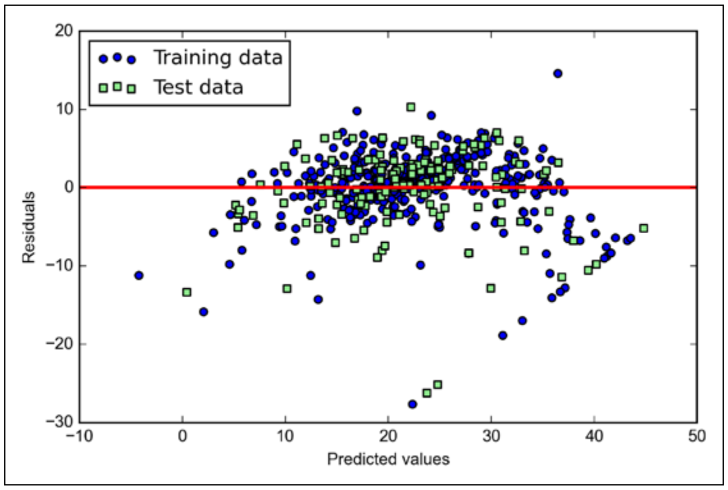
To create a new PythonAnywhere account, we visit the website at `https://www.pythonanywhere.com` and click on the **Pricing & signup** link that is located in the top-right corner. Next, we click on the **Create a Beginner account** button where we need to provide a username, password, and a valid e-mail address. After we have read and agreed to the terms and conditions, we should have a new account.

we
need to create
click

Unfortunately, the free beginner account doesn't allow us to access the remote server via the SSH protocol from our command-line terminal. Thus, we need to use the PythonAnywhere web interface to manage our web application. But before we can upload our local application files to the server, ~~need to create~~ a new web application for our PythonAnywhere account. After we ~~clicking~~ on the **Dashboard** button in the top-right corner, we have access to the control panel shown at the top of the page. Next, we click on the **Web** tab that is now visible at the top of the page. We proceed by clicking on the **Add a new web app** button on the left, which lets us create a new Python 3.4 Flask web application that we name `movieclassifier`.

After creating a new application for our PythonAnywhere account, we head over to the **Files** tab to upload the files from our local `movieclassifier` directory using the PythonAnywhere web interface. After uploading the web application files that we created locally on our computer, we should have a `movieclassifier` directory in our PythonAnywhere account. It contains the same directories and files as our local `movieclassifier` directory has, as shown in the following screenshot:

After executing the code, we should see a residual plot with a line passing through the *x* axis origin as shown here:



In the case of a perfect prediction, the residuals would be exactly zero, which we will probably never encounter in realistic and practical applications. However, for a good regression model, we would expect that the errors are randomly distributed and the residuals should be randomly scattered around the centerline. If we see patterns in a residual plot, it means that our model is unable to capture some explanatory information, which is leaked into the residuals as we can slightly see in our preceding residual plot. Furthermore, we can also use residual plots to detect outliers, which are represented by the points with a large deviation from the centerline.

Another useful quantitative measure of a model's performance is the so-called **Mean Squared Error** (**MSE**), which is simply the average value of the SSE cost function that we minimize to fit the linear regression model. The MSE is useful ~~to~~ for comparing different regression models or for tuning their parameters via a grid search and cross-validation:

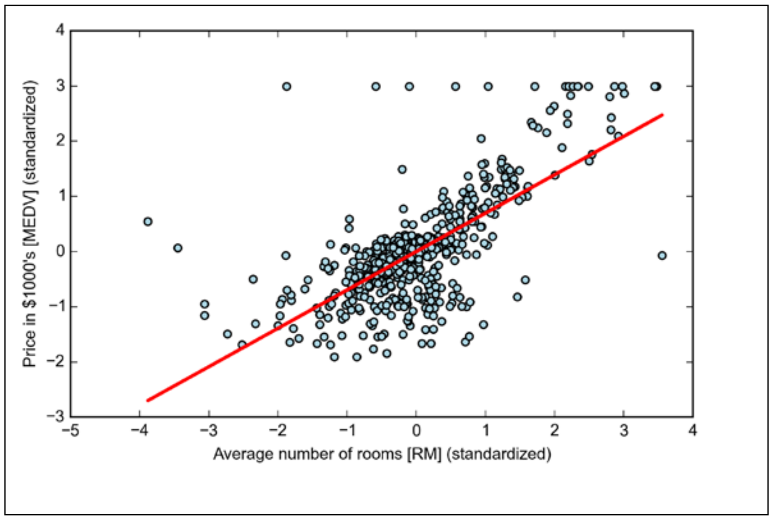
delete "to"

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Execute the following code:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
```

As we can see in the following plot, the linear regression line reflects the general trend that house prices tend to increase with the number of rooms:



Although this observation makes intuitive sense, the data also tells us that the number of rooms does not explain the house prices very well in many cases. Later in this chapter, we will discuss how to quantify the performance of a regression model. Interestingly, we also observe a curious line $y = 3$, which suggests that the prices may have been clipped. In certain applications, it may also be important to report the predicted outcome variables on ~~its~~ original scale. To scale the predicted price outcome back on the **Price in $1000's** axes, we can simply apply the `inverse_transform` method of the `StandardScaler`: their

```
>>> num_rooms_std = sc_x.transform([5.0])
>>> price_std = lr.predict(num_rooms_std)
>>> print("Price in $1000's: %.3f" % \
...         sc_y.inverse_transform(price_std))
Price in $1000's: 10.840
```

In the preceding code example, we used the previously trained linear regression model to predict the price of a house with five rooms. According to our model, such a house is worth $10,840.

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
an 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)
```

by the parameter

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)
```

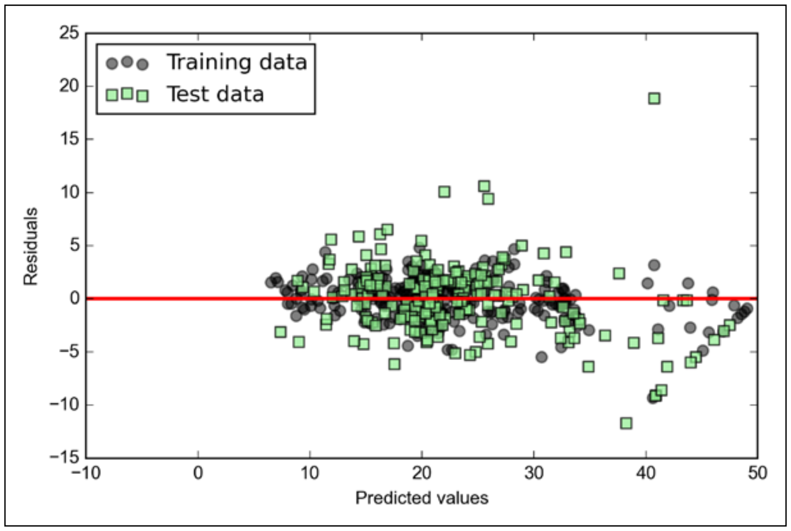
For example, if we set `l1_ratio` to `1.0`, the `ElasticNet` regressor would be equal to LASSO regression. For more detailed information about the different implementations of linear regression, please see the documentation at `http://scikit-learn.org/stable/modules/linear_model.html`.

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:

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

As it was already summarized by the R^2 coefficient, we can see that the model fits the training data better than the test data, as indicated by the outliers in the y axis direction. Also, the distribution of the residuals does not seem to be completely random around the zero center point, indicating that the model is not able to capture all the exploratory information. However, the residual plot indicates a large improvement over the residual plot of the linear model that we plotted earlier in this chapter:

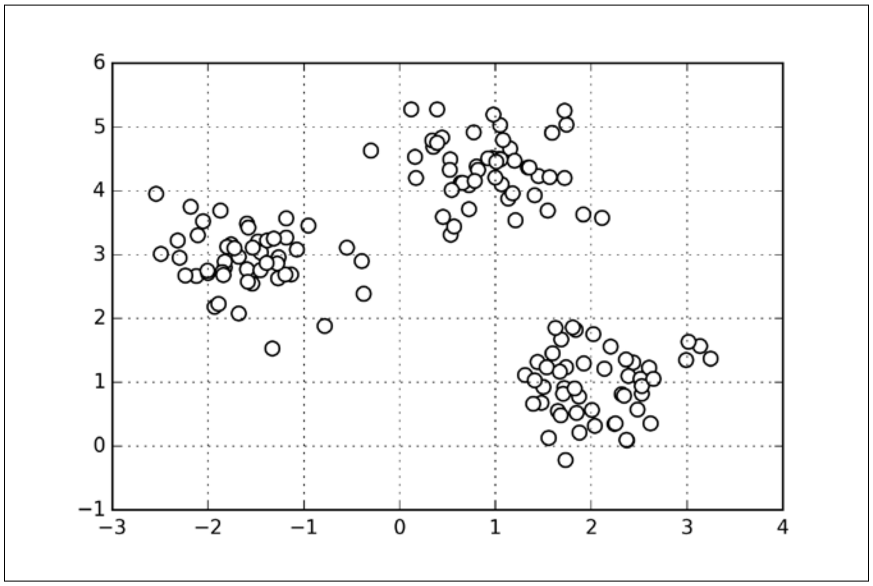


Should also be in italics

In *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, we also discussed the kernel trick that can be used in combination with **support vector machine** (**SVM**) for classification, which is useful if we are dealing with nonlinear problems. Although a discussion is beyond of the scope of this book, SVMs can also be used in nonlinear regression tasks. The interested reader can find more information about Support Vector Machines for regression in an excellent report by S. R. Gunn: S. R. Gunn et al. Support *Vector Machines for Classification and Regression*. (ISIS technical report, 14, 1998). An SVM regressor is also implemented in scikit-learn, and more information about its usage can be found at `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR`.

```
...               marker='o',
...               s=50)
>>> plt.grid()
>>> plt.show()
```

The dataset that we just created consists of 150 randomly generated points that are roughly grouped into three regions with higher density, which is visualized via a two-dimensional scatterplot:



In real-world applications of clustering, we do not have any ground truth category information about those samples; otherwise, it would fall into the category of supervised learning. Thus, our goal is to group the samples based on their feature similarities, which we can be achieved using the k-means algorithm that can be summarized by the following four steps:

1. Randomly pick k centroids from the sample points as initial cluster centers.
2. Assign each sample to the nearest centroid $\mu^{(j)}$, $j \in \{1,\ldots,k\}$.
3. Move the centroids to the center of the samples that were assigned to it.
4. Repeat ~~the~~ steps 2 and 3 until the cluster ~~assignment~~ assignments do not change or a user-defined tolerance or a maximum number of iterations is reached.

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
```

Typically, the approximated difference between the numerical gradient J'_n and analytical gradient J'_a is then calculated as the L2 vector norm. For practical reasons, we unroll the computed gradient matrices into flat vectors so that we can calculate the error (the difference between the gradient vectors) more conveniently:

$$error = \left\| J'_n - J'_a \right\|_2$$

One problem is that the error is not scale invariant (small errors are more significant if the weight vector norms are small too). Thus, it is recommended to calculate a normalized difference:

$$relative\,error = \frac{\left\| J'_n - J'_a \right\|_2}{\left\| J'_n \right\|_2 + \left\| J'_a \right\|_2}$$

Now, we want the relative error between the numerical gradient and the analytical gradient to be as small as possible. Before we implement gradient checking, we need to discuss one more detail: what is the acceptable error threshold to pass the gradient check? The relative error threshold for passing the gradient check depends on the complexity of the network architecture. As a rule of thumb, the more hidden layers we add, the larger the difference between the numerical and analytical gradient can become if backpropagation is implemented correctly. Since we have implemented a relatively simple neural network architecture in this chapter, we want to be rather strict about the threshold and define the following rules:

- Relative error <= 1e-7 means everything is okay!
- Relative error <= 1e-4 means the condition is problematic, and we should look into it.
- Relative error > 1e-4 means there is probably something wrong in our code.

low that we have
established

~~Now we have established~~ these ground rules, let's implement gradient checking. To do so, we can simply take the `NeuralNetMLP` class that we implemented previously and add the following method to the class body:

```
def _gradient_checking(self, X, y_enc, w1,
                       w2, epsilon, grad1, grad2):
    """ Apply gradient checking (for debugging only)

    Returns
    ---------
```

What is Theano?

What exactly is Theano—a programming language, a compiler, or a Python library? It turns out that it fits all these descriptions. Theano has been developed to implement, compile, and evaluate mathematical expressions very efficiently with a strong focus on multidimensional arrays (tensors). It comes with an option to run code on CPU(s). However, its real power comes from utilizing GPUs to take advantage of the large memory bandwidths and great capabilities for floating point math. Using Theano, we can easily run code in parallel over shared memory as well. In 2010, the developers of Theano reported an 1.8x faster performance than NumPy when the code was run on the CPU, and if Theano targeted the GPU, it was even 11x faster than NumPy (J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: *A CPU and GPU Math Compiler in Python*. In Proc. 9th Python in Science Conf, pages 1–7, 2010.). Now, keep in mind that this benchmark is from 2010, and Theano has improved significantly over the years, and so have the capabilities of modern graphics cards.

"Theano" should also be in italics

So, how does Theano relate to NumPy? Theano is built on top of NumPy and it has a very similar syntax, which makes the usage very convenient for people who are already familiar with the latter. To be fair, Theano is not just "NumPy on steroids" as many people would describe it, but it also shares some similarities with SymPy (`http://www.sympy.org`), a Python package for symbolic computations (or symbolic algebra). As we saw in previous chapters, in NumPy, we describe what our variables are, and how we want to combine them; then, the code is executed line by line. In Theano, however, we write down the problem first and the description of how we want to analyze it. Then, Theano optimizes and compiles code for us using C/C++, or CUDA/OpenCL if we want to run it on the GPU. In order to generate the optimized code for us, Theano needs to know the scope of our problem; think of it as a tree of operations (or a graph of symbolic expressions). Note that Theano is still under active development, and many new features are added and improvements are made on a regular basis. In this chapter, we will explore the basic concepts behind Theano and learn how to use it for machine learning tasks. Since Theano is a large library with many advanced features, it would be impossible to cover all of them in this book. However, I will provide useful links to the excellent online documentation (`http://deeplearning.net/software/theano/`) if you want to learn more about this library.

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

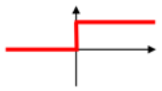
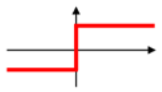
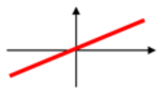
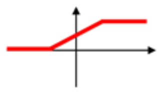
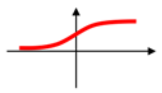
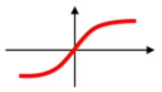
The logistic activation function that we used in the previous chapter probably mimics the concept of a neuron in a brain most closely: we can think of it as probability of whether a neuron fires or not. However, logistic activation functions can be problematic if we have highly negative inputs, since 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.

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()
```

Now that we know more about the different activation functions that are commonly used in artificial neural networks, let's conclude this section with an overview of the different activation ~~function~~ that we encountered in this book.

“functions” (plural)

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z)=\begin{cases}0, & z<0,\\ 0.5, & z=0,\\ 1, & z>0,\end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z)=\begin{cases}-1, & z<0,\\ 0, & z=0,\\ 1, & z>0,\end{cases}$	Perceptron variant	
Linear	$\phi(z)=z$	Adaline, linear regression	
Piece-wise linear	$\phi(z)=\begin{cases}1, & z\geq\frac{1}{2},\\ z+\frac{1}{2}, & -\frac{1}{2}<z<\frac{1}{2},\\ 0, & z\leq-\frac{1}{2},\end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z)=\frac{1}{1+e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	Multi-layer NN	

Training neural networks efficiently using Keras

In this section, we will take a look at Keras, one of the most recently developed libraries to facilitate neural network training. The development on Keras started in the early months of 2015; as of today, it has evolved into one of the most popular and widely used libraries that are built on top of Theano, and allows us to utilize our GPU to accelerate neural network training. One of its prominent features is that it's a very intuitive API, which allows us to implement neural networks in only a few lines of code. Once you have Theano installed, you can install Keras from PyPI by executing the following command from your terminal command line:

```
pip install Keras
```

For more information about Keras, please visit the official website at `http://keras.io`.

To see what neural network training via Keras looks like, let's implement a multilayer perceptron to classify the handwritten digits from the MNIST dataset, which we introduced in the previous chapter. The MNIST dataset can be downloaded from `http://yann.lecun.com/exdb/mnist/` in four parts as listed here:

- `train-images-idx3-ubyte.gz`: These are training set images (9912422 bytes)
- `train-labels-idx1-ubyte.gz`: These are training set labels (28881 bytes)
- `t10k-images-idx3-ubyte.gz`: These are test set images (1648877 bytes)
- `t10k-labels-idx1-ubyte.gz`: These are test set labels (4542 bytes)

unzipping After downloading and ~~unzipped~~ the archives, we place the files into a directory `mnist` in our current working directory, so that we can load the training as well as the test dataset using the following function:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                                % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                               % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                             dtype=np.uint8).reshape(len(labels), 784)
```

Printing the value of the cost function is extremely useful during training, since we can quickly spot whether the cost is decreasing during training and stop the algorithm earlier if otherwise to tune the hyperparameter~~s~~ values. delete extra “s”

To predict the class labels, we can then use the `predict_classes` method to return the class labels directly as integers:

```
>>> y_train_pred = model.predict_classes(X_train, verbose=0)
>>> print('First 3 predictions: ', y_train_pred[:3])
>>> First 3 predictions:  [5 0 4]
```

Finally, let's print the model accuracy on training and test sets:

```
>>> train_acc = np.sum(
...       y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (train_acc * 100))
Training accuracy: 94.51%

>>> y_test_pred = model.predict_classes(X_test, verbose=0)
>>> test_acc = np.sum(y_test == y_test_pred,
...                   axis=0) / X_test.shape[0]
print('Test accuracy: %.2f%%' % (test_acc * 100))
Test accuracy: 94.39%
```

Note that this is just a very simple neural network without optimized tuning parameters. If you are interested in playing more with Keras, please feel free to further tweak the learning rate, momentum, weight decay, and number of hidden units.

a

Although Keras is great library for implementing and experimenting with neural networks, there are many other Theano wrapper libraries that are worth mentioning. A prominent example is Pylearn2 (`http://deeplearning.net/software/pylearn2/`), which has been developed in the LISA lab in Montreal. Also, Lasagne (`https://github.com/Lasagne/Lasagne`) may be of interest to you if you prefer a more minimalistic but extensible library, that offers more control over the underlying Theano code.

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~~ you there! meeting
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.