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

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

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

What this book covers

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

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

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

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

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

What you need for this book

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

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

Who this book is for

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

to

Conventions

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

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

More formally, we can pose this problem as a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. We can then define an *activation function* $\phi(z)$ that takes a linear combination of certain input values $\boldsymbol{x}$ and a corresponding weight vector $\boldsymbol{w}$, where z is the so-called net input ($z = w_1 x_1 + \ldots + w_m x_m$):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

> If you are not yet familiar with Python's scientific libraries or need a refresher, please see the following resources:
>
> **NumPy**: `http://wiki.scipy.org/Tentative_NumPy_Tutorial`
>
> **Pandas**: `http://pandas.pydata.org/pandas-docs/stable/tutorials.html`
>
> **Matplotlib**: `http://matplotlib.org/ussers/beginner.html`
>
> Also, to better follow the code examples, I recommend you download the IPython notebooks from the Packt website. For a general introduction to IPython notebooks, please visit `https://ipython.org/ipython-doc/3/notebook/index.html`.

users

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

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

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

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

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

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples
            is the number of samples and
```
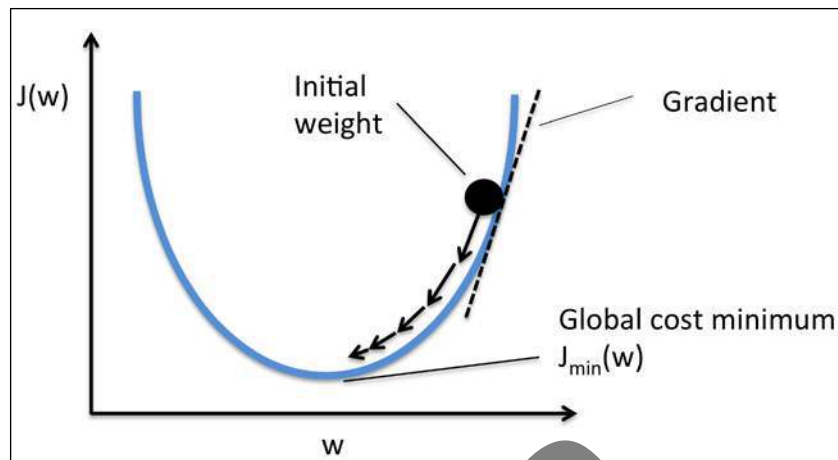
Using gradient descent, we can now update the weights by taking a step away from the gradient $\nabla J(w)$ of our cost function $J(w)$:

$$w := w + \Delta w$$

Here, the weight change Δw is defined as the negative gradient multiplied by the learning rate η:

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

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 $w := w + \Delta w$.

η (Greek "eta")

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:

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

$$\prod_{i=1}^{n}$$

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

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

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*:

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

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:

$$J(\phi(z), y; w) = -y \log(\phi(z)) - (1 - y)\log(1 - \phi(z))$$

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

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

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

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

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

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

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

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

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

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

$$= \left(y - \phi(z) \right) x_j$$

Maximizing information gain – getting the most bang for the buck

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

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

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

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

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

Gini impurity

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

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

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

(1 - p(i | t))

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

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

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

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

Another impurity measure is the classification error:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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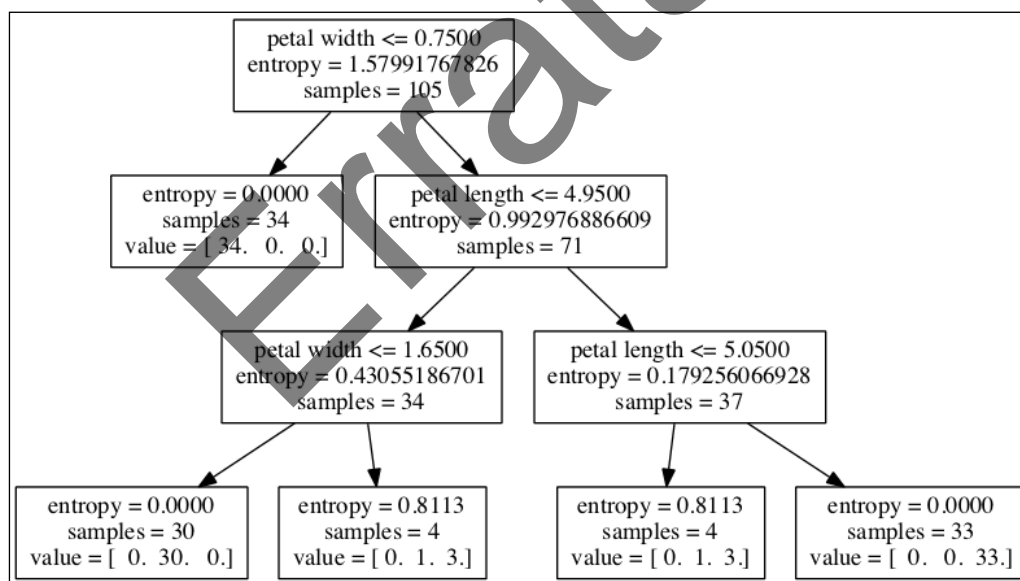

Looking at the decision tree figure that we created via GraphViz, we can now nicely trace back the splits that the decision tree determined from our training dataset. We started with 105 samples at the root and split it into two child nodes with 34 and 71 samples each using the **petal with** cut-off ≤ 0.75 cm. After the first split, we can see that the left child node is already pure and only contains samples from the Iris-Setosa class (entropy = 0). The further splits on the right are then used to separate the samples from the Iris-Versicolor and Iris-Virginica classes.
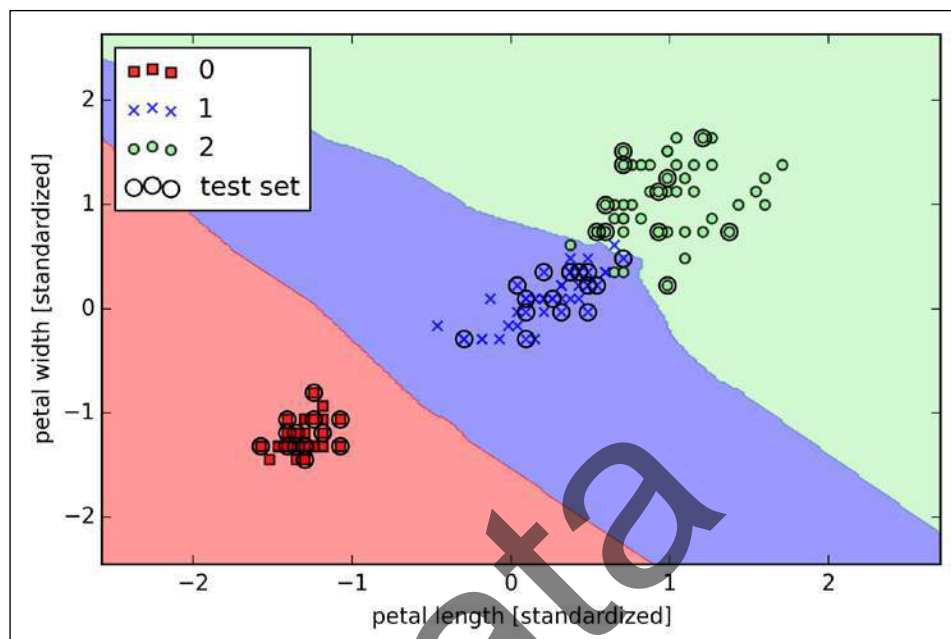
In the case of a tie, the scikit-learn implementation of the KNN algorithm will prefer the neighbors with a closer distance to the sample. If the neighbors have a similar distance, the algorithm will choose the class label that comes first in the training dataset.

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

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

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

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

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

$$\sum v = \lambda v$$

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

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

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

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

Insert
this
note

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

Compressing Data via Dimensionality Reduction

Replace text with

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

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

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

As we can see in the resulting figure, the first two linear discriminants capture about 100 percent of the useful information in the *Wine* training dataset:
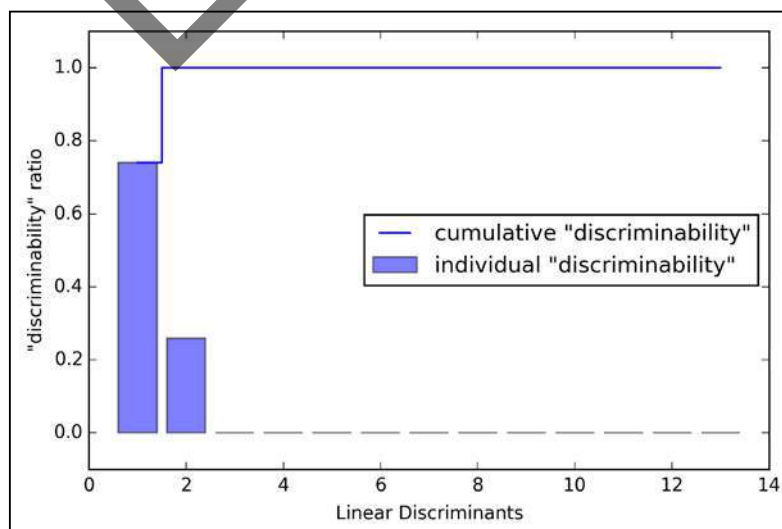
We do this for each pair of samples:

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

(n)

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

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

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

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

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

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

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

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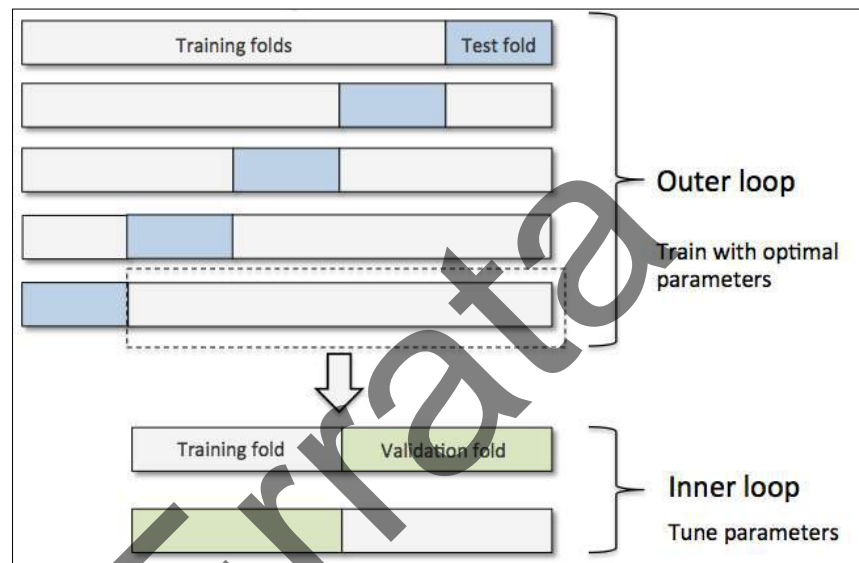
In scikit-learn, we can perform nested cross-validation as follows:

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

remove indent

cv= 5,

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

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

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

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

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

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

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

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

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

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

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

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

Exploring the Housing Dataset

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

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

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

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

$$\mathbf{w} = (X^T X)^{-1} X^T y$$
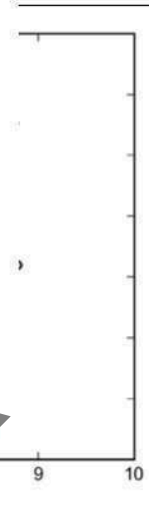
We can implement it in Python as follows:

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

Slope: 9.102

>>> print('Intercept: %.3f' % w[0])

Intercept: -34.671
```

executing the code GD implementation:

Average number of rooms [RM]

replace section with...

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

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

$$w_0 = \mu_y - \mu_{\hat{y}} \mu_{\hat{y}}$$

Here, μ_y is the mean of the true target values and $\mu_{\hat{y}}$ is the mean of the predicted response.

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

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

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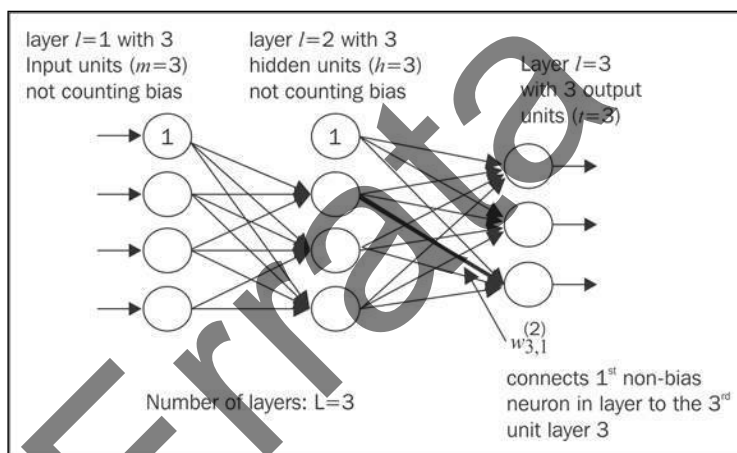
Activating a neural network via forward propagation

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

1. Starting at the input layer, we forward propagate the patterns of the training data through the network to generate an output.

2. Based on the network's output, we calculate the error that we want to minimize using a cost function that we will describe later.

3. We backpropagate the error, find its derivative with respect to each weight in the network, and update the model.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

j=1

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

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

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

Concisely, we just forward propagate the input features through the connection in the network as shown here:
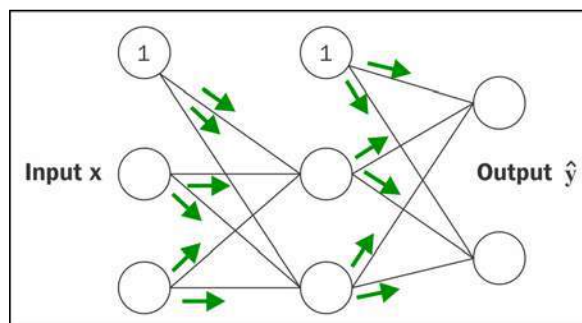
In backpropagation, we propagate the error from right to left. We start by calculating the error vector of the output layer:

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

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

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

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

$$\delta^{(3)}$$

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:

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

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