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 ($z = w_1 x_1 + \ldots + w_m x_m$):

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

**net input z**

Now, ~~if the activation~~ of a particular sample $x^{(i)}$~~, that is, the output of $\phi(z)$~~, is greater than a defined threshold $\theta$, 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*:

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

Where $\eta$ 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 $\Delta w_j$ were updated. Concretely, for a 2D dataset, we would write the update as follows:

Please note that the learning rate η (eta) only has an effect on the classification outcome if the weights are initialized to non-zero values. If all the weights are initialized to 0, only the scale of the weight vector, not the direction. To have the learning rate influence the classification outcome, the weights need to be initialized to non-zero values. The respective lines in the code that need to be changed to accomplish that are highlighted on below:

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

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

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

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:

```
def __init__(self, eta=0.01, n_iter=50, random_seed=1):
    ...
    self.random_seed = random_seed

 def fit(self, X, y):
    ...
~~self.w_ = np.zeros(1 + X.shape[1])~~
    rgen = np.random.RandomState(self.random_seed)
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
```

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

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

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

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

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

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

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

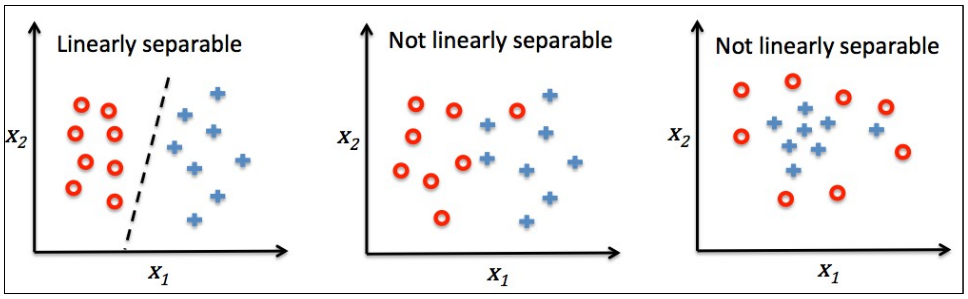
Let's assume that $x_j^{(i)} = 0.5$, and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1 so that the net input ~~activation~~ $x_j^{(i)} \times w_j^{\cancel{(i)}}$ will be more positive the next time we encounter this sample and thus will be more likely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

$$\Delta w_{\cancel{i}\,j}^{\cancel{(i)}} = (1 - -1)0.5 = (2)0.5 = 1$$

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 extent to classify this sample correctly the next time:

$$\Delta w_j = (1 - -1)2 = (2)2 = 4$$

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



**Downloading the example code**

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

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

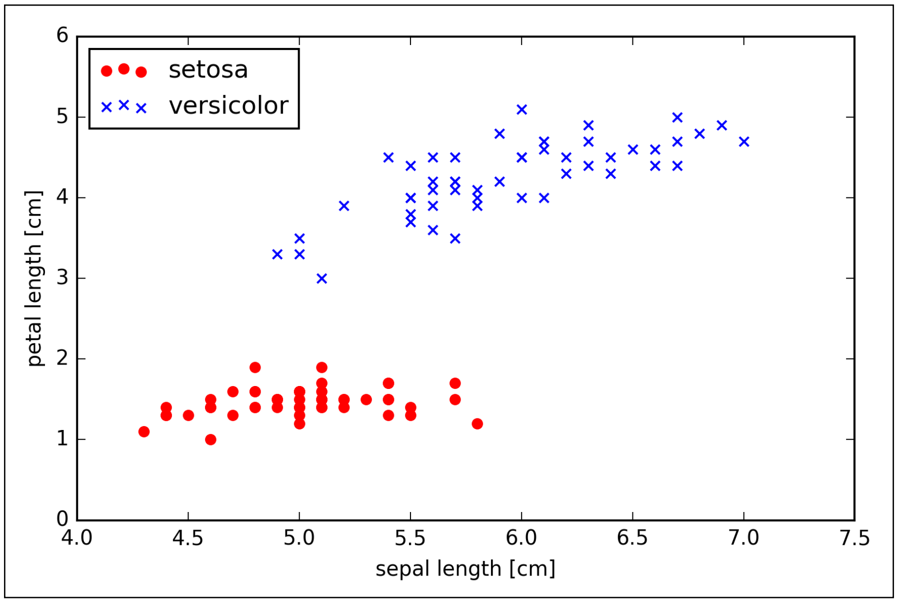
Instead of using NumPy to calculate the vector dot product between two arrays a and b via `a.dot(b)` or `np.dot(a, b)`, we could also perform the calculation in pure Python via `sum([i*j for i,j in zip(a, b)])`. However, the advantage of using NumPy over classic Python for-loop structures is that its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

## Training a perceptron model on the Iris dataset

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

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

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



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

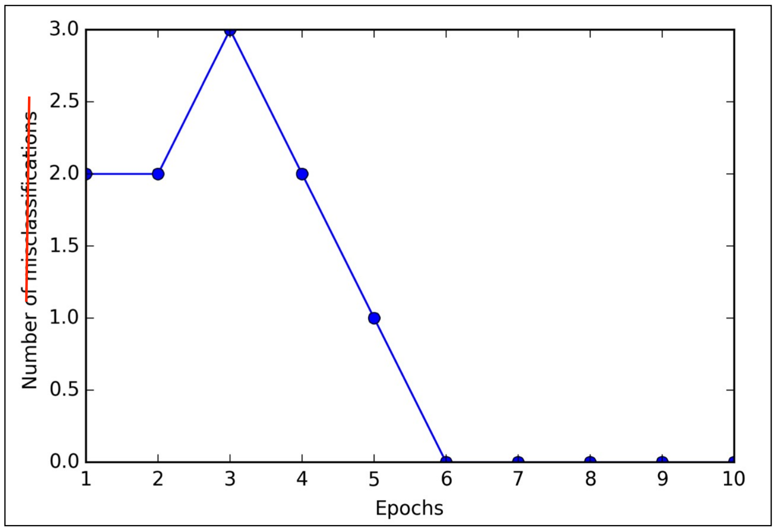
```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_,
```

Note: Although “misclassifications” isn’t technically wrong in this context (referring to the number of misclassified samples while updating), it could be easily confused with the number of misclassifications on the training set (after the perceptron model was fit). So, the label “number of updates” (vs. “number of misclassifications”) is hopefully less ambiguous.

```
...          marker='o')
>>> plt.xlabel('Epochs')          updates
>>> plt.ylabel('Number of ~~misclassifications~~')
>>> plt.show()
```

After executing the preceding code, we should see the plot of the misclassification errors versus the number of epochs, as shown next:

updates



As we can see in the preceding plot, our perceptron already converged after the sixth epoch and should now be able to classify the training samples perfectly. Let us implement a small convenience function to visualize the decision boundaries for 2D datasets:

```
from matplotlib.colors import ListedColormap


def plot_decision_regions(X, y, classifier, resolution=0.02):


    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

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

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

The `plt.scatter` function in the `plot_decision_regions` plot may raise errors if you have matplotlib <= 1.5.0 installed if you use this function to plot more than four classes as a reader pointed out: "[...] if there are four items to be displayed as the RGBA tuple is mis-interpreted as a list of colours".

```python
plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
          alpha=0.8, c=cmap(idx),
          marker=markers[idx], label=cl)
```

To address this problem in older matplotlib versions, you can replace `c=cmap(idx)` by `c=colors[idx]`.

First, we define a number of `colors` and `markers` and create a color map from the list of colors via `ListedColormap`. Then, we determine the minimum and maximum values for the two features and use those feature vectors to create a pair of grid arrays `xx1` and `xx2` via the NumPy `meshgrid` function. Since we trained our perceptron classifier on two feature dimensions, we need to flatten the grid arrays and create a matrix that has the same number of columns as the Iris training subset so that we can use the `predict` method to predict the class labels `Z` of the corresponding grid points. After reshaping the predicted class labels `Z` into a grid with the same dimensions as `xx1` and `xx2`, we can now draw a contour plot via matplotlib's `contourf` function that maps the different decision regions to different colors for each predicted class in the grid array:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

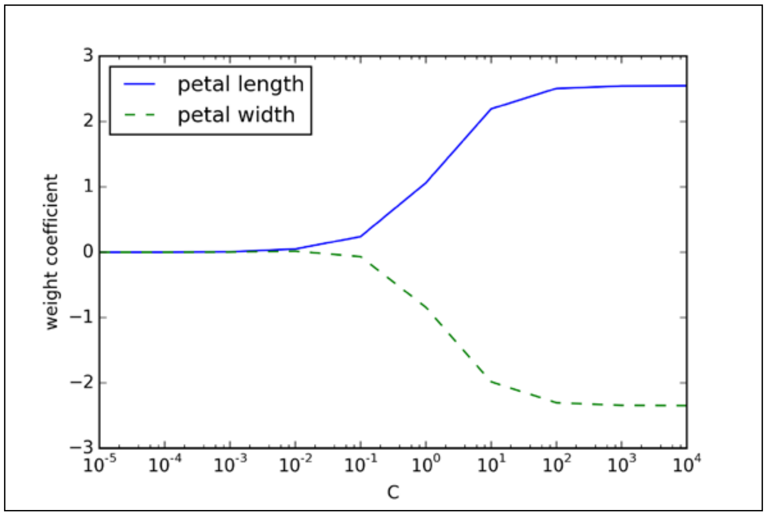
Consequently, decreasing the value of the inverse regularization parameter `C` means that we are increasing the regularization strength, which we can visualize by plotting the L2 regularization path for the two weight coefficients:

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

Since the class labels in scikit-learn's iris are {0, 1, 2}, this should be "the '2nd' class in the dataset (sorted in ascending order), which refers to class label 1"

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

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



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

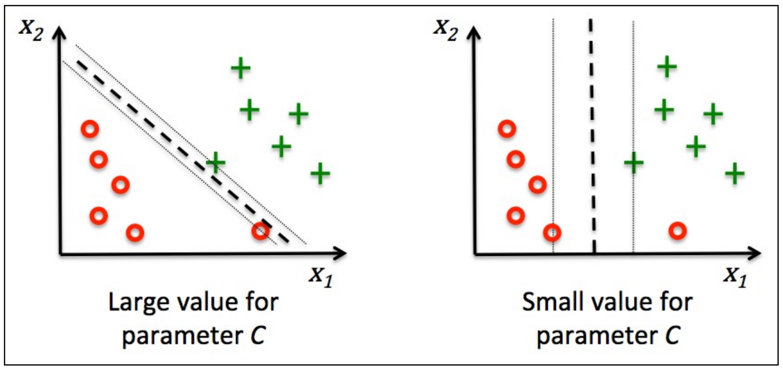
$$\boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 - \xi^{(i)} \; if \;\; y^{(i)} = 1$$

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

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

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

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



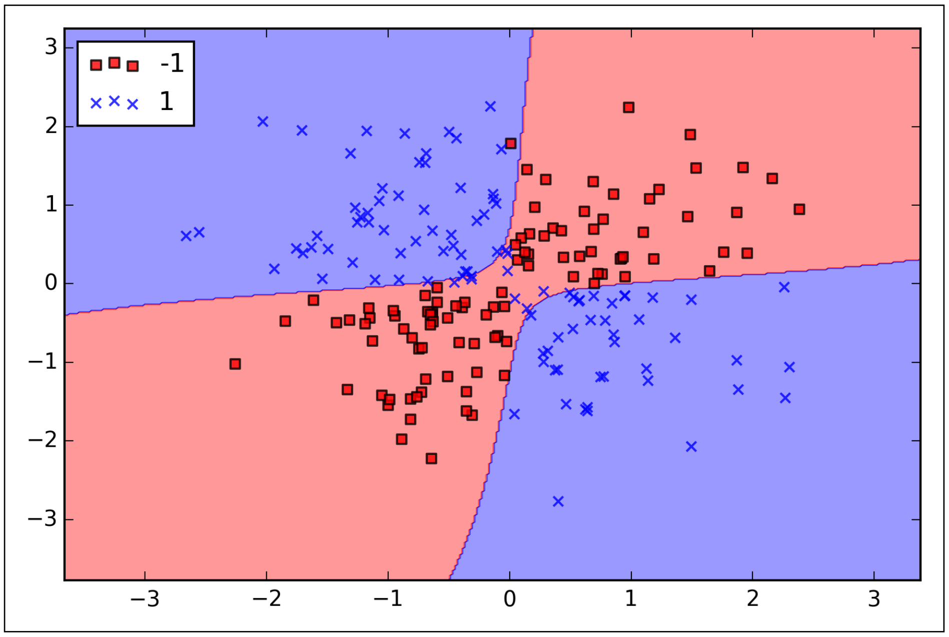
This concept is related to regularization, which we discussed previously in the context of regularized regression where increasing the value of `C` increases the bias and lowers the variance of the model.

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

Now that we defined the big picture behind the kernel trick, let's see if we can train a kernel SVM that is able to draw a nonlinear decision boundary that separates the XOR data well. Here, we simply use the `SVC` class from scikit-learn that we imported earlier and replace the parameter `kernel='linear'` with `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.show()
```

As we can see in the resulting plot, the kernel SVM separates the XOR data relatively well:



tighter, bumpier or more pointed

The $\gamma$ parameter, which we set to `gamma=0.1`, can be understood as a *cut-off* parameter for the Gaussian sphere. If we increase the value for $\gamma$, we increase the influence or reach of the training samples, which leads to a ~~softer~~ decision boundary. To get a better intuition for $\gamma$, let's apply RBF kernel SVM to our Iris flower dataset:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
```

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

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

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

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

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

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

Another impurity measure is the classification error:

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

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

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

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

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

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

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

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

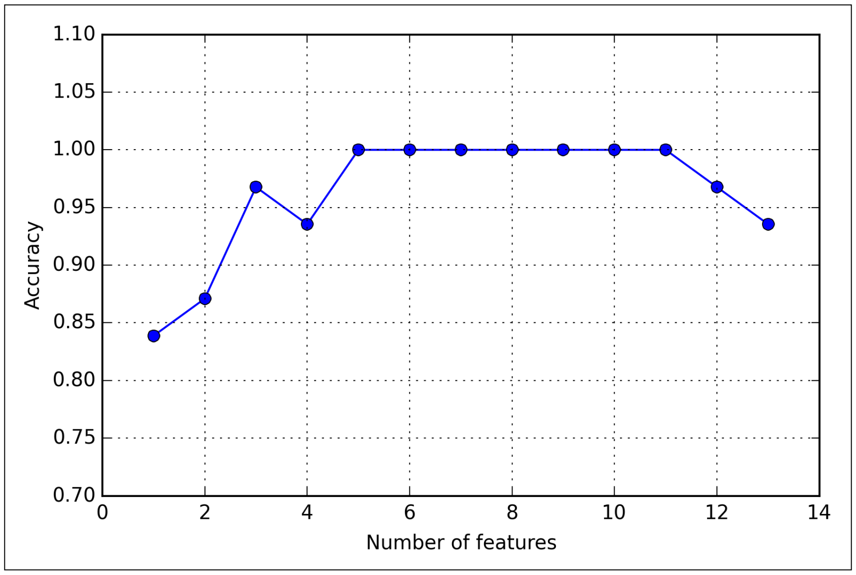
$$z = w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}$$

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

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

```
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Number of features')
>>> plt.grid()
>>> plt.show()
```

As we can see in the following plot, the accuracy of the KNN classifier improved on the validation dataset as we reduced the number of features, which is likely due to a decrease of **the curse of dimensionality** that we discussed in the context of the KNN algorithm in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*. Also, we can see in the following plot that the classifier achieved 100 percent accuracy for *k={5, 6, 7, 8, 9, 10}*: include "11" in the list



To satisfy our own curiosity, let's see what those five features are that yielded such a good performance on the validation dataset:

```
>>> k5 = list(sbs.subsets_[8])
>>> print(df_wine.columns[1:][k5])
Index(['Alcohol', 'Malic acid', 'Alcalinity of ash', 'Hue',
'Proline'], dtype='object')
```

Using the preceding code, we obtained the column indices of the 5-feature subset from the 9th position in the `sbs.subsets_` attribute and returned the corresponding feature names from the column-index of the pandas Wine `DataFrame`.

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

## Feature transformation

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

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

missing whtespace between "in" and "range"

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

**eigen_pairs.sort(key=lambda k: k[0], reverse=True)**

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

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

**Note: I added the `key=lambda k: k[0]` in the sort call above just like I used it further below in the LDA section. This is to avoid problems if there are ties in the eigenvalue arrays (i.e., the sorting algorithm will only regard the first element of the tuples, now).**

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

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

## Computing the scatter matrices

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

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

This results in three mean vectors:

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
[[ 0.0662 -0.3797]
[-0.0386 -0.2206]
[ 0.0217 -0.3816]
[-0.184 0.3018]
[ 0.0034 0.0141]
[-0.2326 0.0234]
[ 0.7747 0.1869]
[ 0.0811 0.0696]
[-0.0875 0.1796]
[-0.185 -0.284 ]
[ 0.066 0.2349]
[ 0.3805 0.073 ]
[ 0.3285 -0.5971]]
```

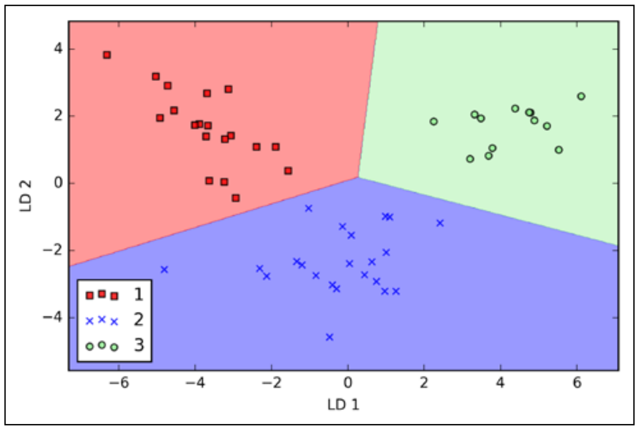
## Projecting samples onto the new feature space

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

$$\boldsymbol{X}' = \boldsymbol{XW}$$

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

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

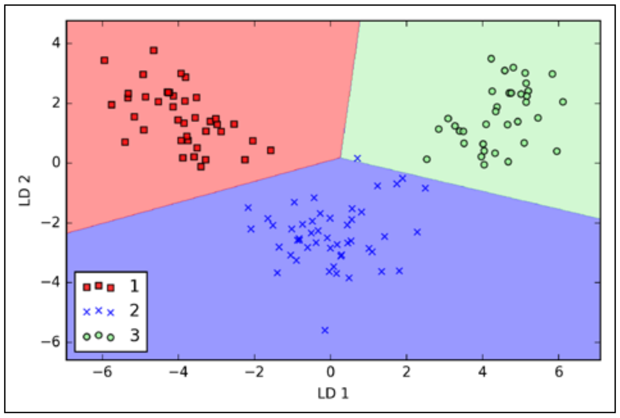


By lowering the regularization strength, we could probably shift the decision boundaries so that the logistic regression models classify all samples in the training dataset correctly. However, let's take a look at the results on the test set:

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

For some reason, these 2 images got flipped during layouting

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



Note that the preceding equation refers to the covariance between two features; now, let's write the general equation to calculate the covariance *matrix* $\Sigma$:

$$\sum = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}^{(i)} \, \boldsymbol{x}^{(i)T}$$

Bernhard Scholkopf generalized this approach (B. Scholkopf, A. Smola, and K.-R. Muller. *Kernel Principal Component Analysis*. pages 583–588, 1997) so that we can replace the dot products between samples in the original feature space by the nonlinear feature combinations via $\phi$:

$$\sum = \frac{1}{n} \sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right) \phi(\boldsymbol{x}^{(i)})^{T}$$

To obtain the eigenvectors—the principal components—from this covariance matrix, we have to solve the following equation:

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

$$\Rightarrow \frac{1}{n} \sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right) \phi\left(\boldsymbol{x}^{(i)}\right)^{T} \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow v = \frac{1}{n\lambda} \sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right) \phi\left(\boldsymbol{x}^{(i)}\right)^{T} \boldsymbol{v} = \cancel{\frac{1}{n}} \sum_{i=1}^{n} \boldsymbol{a}^{(i)} \phi\left(\boldsymbol{x}^{(i)}\right)$$

Here, $\lambda$ and $\boldsymbol{v}$ are the eigenvalues and eigenvectors of the covariance matrix $\Sigma$, and $\boldsymbol{a}$ can be obtained by extracting the eigenvectors of the kernel (similarity) matrix $\boldsymbol{K}$ as we will see in the following paragraphs.

The derivation of the kernel matrix is as follows:

First, let's write the covariance matrix as in matrix notation, where $\phi(X)$ is an $n \times k$ -dimensional matrix:

$$\sum = \frac{1}{n} \sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right) \phi\left(\boldsymbol{x}^{(i)}\right)^{T} = \frac{1}{n} \phi(\boldsymbol{X})^{T} \phi(\boldsymbol{X})$$

Now, we can write the eigenvector equation as follows:

$$v = \cancel{\frac{1}{n}} \sum_{i=1}^{n} a^{(i)} \phi\left(\boldsymbol{x}^{(i)}\right) = \cancel{\lambda} \phi\left(\boldsymbol{X}\right)^T \boldsymbol{a}$$

Since $\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$, we get:

$$\frac{1}{n} \phi(\boldsymbol{X})^T \phi(\boldsymbol{X}) \phi(\boldsymbol{X})^T \boldsymbol{a} = \lambda \phi(\boldsymbol{X})^T \boldsymbol{a}$$

Multiplying it by $\phi(\boldsymbol{X})$ on both sides yields the following result:

$$\frac{1}{n} \phi(\boldsymbol{X}) \phi(\boldsymbol{X})^T \phi(\boldsymbol{X}) \phi(\boldsymbol{X})^T \boldsymbol{a} = \lambda \phi(\boldsymbol{X}) \phi(\boldsymbol{X})^T \boldsymbol{a}$$

$$\Rightarrow \frac{1}{n} \phi(\boldsymbol{X}) \phi(\boldsymbol{X})^T \boldsymbol{a} = \lambda \boldsymbol{a}$$

$$\Rightarrow \frac{1}{n} \boldsymbol{K} \boldsymbol{a} = \lambda \boldsymbol{a}$$

Here, $\boldsymbol{K}$ is the similarity (kernel) matrix:

$$\boldsymbol{K} = \phi(\boldsymbol{X}) \phi(\boldsymbol{X})^T$$

As we recall from the SVM section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we use the kernel trick to avoid calculating the pairwise dot products of the samples $\boldsymbol{x}$ under $\phi$ explicitly by using a kernel function $\boldsymbol{K}$ so that we don't need to calculate the eigenvectors explicitly:

$$k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^T \phi\left(\boldsymbol{x}^{(j)}\right)$$

In other words, what we obtain after kernel PCA are the samples already projected onto the respective components rather than constructing a transformation matrix as in the standard PCA approach. Basically, the kernel function (or simply *kernel*) can be understood as a function that calculates a dot product between two vectors—a measure of similarity.

To predict a class label via a simple majority or plurality voting, we combine the predicted class labels of each individual classifier $C_j$ and select the class label $\hat{y}$ that received the most votes:

$$\hat{y} = mode\left\{ C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x}) \right\}$$

For example, in a binary classification task where $class1 = -1$ and $class2 = +1$, we can write the majority vote prediction as follows:

$$C(\boldsymbol{x}) = sign\left[ \sum_{j}^{m} C_j(\boldsymbol{x}) \right] = \begin{cases} 1 & if \sum_i C_j(\boldsymbol{x}) \geq 0 \\ -1 & otherwise \end{cases}$$

To illustrate why ensemble methods can work better than individual classifiers alone, let's apply the simple concepts of combinatorics. For the following example, we make the assumption that all $n$ base classifiers for a binary classification task have an equal error rate $\varepsilon$. Furthermore, we assume that the classifiers are independent and the error rates are not correlated. Under those assumptions, we can simply express the error probability of an ensemble of base classifiers as a probability mass function of a binomial distribution:

$$P(y \geq k) = \sum_{k}^{n} \left\langle \begin{matrix} n \\ k \end{matrix} \right\rangle \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{ensemble}$$

Here, $\left\langle \begin{matrix} n \\ k \end{matrix} \right\rangle$ is the binomial coefficient $n$ *choose* $k$. In other words, we compute the probability that the prediction of the ensemble is wrong. Now let's take a look at a more concrete example of 11 base classifiers ($n = 11$) with an error rate of 0.25 ($\varepsilon = 0.25$):

$$P(y \geq k) = \sum_{k=6}^{11} \left\langle \begin{matrix} 11 \\ k \end{matrix} \right\rangle 0.25^k \left(1 - \underset{\substack{\downarrow \\ 0.25}}{\varepsilon}\right)^{11-k} = 0.034$$

As we can see, the error rate of the ensemble (0.034) is much lower than the error rate of each individual classifier (0.25) if all the assumptions are met. Note that, in this simplified illustration, a 50-50 split by an even number of classifiers $n$ is treated as an error, whereas this is only true half of the time. To compare such an idealistic ensemble classifier to a base classifier over a range of different base error rates, let's implement the probability mass function in Python:

```
>>> from scipy.misc import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = int(math.ceil(n_classifier / 2.0))
...     probs = [comb(n_classifier, k) *
...              error**k *
...              (1-error)**(n_classifier - k)
...              for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.034327507019042969
```

For historical reasons, Python 2.7's `math.ceil` returns a `float` instead of an integer like in Python 3.x. Although Although this book was written for Python >3.4, let's make it compatible to Python 2.7 by casting it to an it explicitely:

After we've implemented the `ensemble_error` function, we can compute the ensemble error rates for a range of different base errors from 0.0 to 1.0 to visualize the relationship between ensemble and base errors in a line graph:

```
>>> import numpy as np
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> import matplotlib.pyplot as plt
>>> plt.plot(error_range, ens_errors,
...          label='Ensemble error',
...          linewidth=2)
>>> plt.plot(error_range, error_range,
...          linestyle='--', label='Base error',
...          linewidth=2)
>>> plt.xlabel('Base error')
>>> plt.ylabel('Base/Ensemble error')
>>> plt.legend(loc='upper left')
>>> plt.grid()
>>> plt.show()
```

As we can see in the resulting plot, the error probability of an ensemble is always better than the error of an individual base classifier as long as the base classifiers perform better than random guessing ($\varepsilon < 0.5$). Note that the $y$-axis depicts the base error (dotted line) as well as the ensemble error (continuous line):

To translate the concept of the weighted majority vote into Python code, we can use NumPy's convenient `argmax` and `bincount` functions:

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...              weights=[0.2, 0.2, 0.6]))
1
```

As discussed in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, certain classifiers in scikit-learn can also return the probability of a predicted class label via the `predict_proba` method. Using the predicted class probabilities instead of the class labels for majority voting can be useful if the classifiers in our ensemble are well calibrated. The modified version of the majority vote for predicting class labels from probabilities can be written as follows:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$$

Here, $p_{ij}$ is the predicted probability of the *jth* classifier for class label *i*.

To continue with our previous example, let's assume that we have a binary classification problem with class labels $i \in \{0,1\}$ and an ensemble of three classifiers $C_j$ ($j \in \{1,2,3\}$). Let's assume that the classifier $C_j$ returns the following class membership probabilities for a particular sample $\boldsymbol{x}$:

$$C_1(\boldsymbol{x}) \to [0.9, 0.1],\ C_2(\boldsymbol{x}) \to [0.8, 0.2],\ C_3(\boldsymbol{x}) \to [0.4, 0.6]$$

We can then calculate the individual class probabilities as follows:

$$p(i_0 \mid \boldsymbol{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1 \mid \boldsymbol{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.\cancel{0}6 = 0.42$$

$$\hat{y} = \arg\max_i \left[ p(i_0 \mid \boldsymbol{x}), p(i_1 \mid \boldsymbol{x}) \right] = 0$$

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

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

Layout issue. This was supposed to be indented for better clarity

```
1. Set weight vector $\boldsymbol{w}$ to uniform weights where $\sum_i w_i = 1$
2. For $j$ in $m$ boosting rounds, do the following:
3. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
4. Predict class labels: $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$.
5. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq \boldsymbol{y})$. “not equal” sign
6. Compute coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
7. Update weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
8. Normalize weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
9. Compute final prediction: $\hat{\boldsymbol{y}} = \left( \sum_{j=1}^{m} \left( \boldsymbol{\alpha}_j \times \text{predict}(C_j, \boldsymbol{X}) \right) > 0 \right)$.
```

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

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

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=~~None,~~
...                               random_state=0)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=0)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/testText accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.845/0.854
```

That's a typo (or copy & paste failure). The actual results were correctly produced by setting `max_depth=1` since we are "boosting" weak learners (i.e., decision tree stumps)

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

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

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

## Introducing the bag-of-words model

We remember from *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, that we have to convert categorical data, such as text or words, into a numerical form before we can pass it on to a machine learning algorithm. In this section, we will introduce the **bag-of-words** model that allows us to represent text as numerical feature vectors. The idea behind the bag-of-words model is quite simple and can be summarized as follows:

1. We create a **vocabulary** of unique **tokens**—for example, words—from the entire set of documents.
2. We construct a feature vector from each document that contains the counts of how often each word occurs in the particular document.

Since the unique words in each document represent only a small subset of all the words in the bag-of-words vocabulary, the feature vectors will consist of mostly zeros, which is why we call them **sparse**. Do not worry if this sounds too abstract; in the following subsections, we will walk through the process of creating a simple bag-of-words model step-by-step.

## Transforming words into feature vectors

To construct a bag-of-words model based on the word counts in the respective documents, we can use the `CountVectorizer` class implemented in scikit-learn. As we will see in the following code section, the `CountVectorizer` class takes an array of text data, which can be documents or just sentences, and constructs the bag-of-words model for us:

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array([
...         'The sun is shining',
...         'The weather is sweet',
...         'The sun is shining and the weather is sweet'])
>>> bag = count.fit_transform(docs)
```

By calling the `fit_transform` method on `CountVectorizer`, we just constructed the vocabulary of the bag-of-words model and transformed the following three sentences into sparse feature vectors:

1. `The sun is shining`
2. `The weather is sweet`
3. ~~`The sun is shining and the weather is sweet`~~
   **The sun is shining, the weather is sweet, and one and one is two**

**[A better example suggested by Claude Coulombe]**

Now let us print the contents of the vocabulary to get a better understanding of the underlying concepts:

```
    >>> print(count.vocabulary_)
    ~~{'the': 5, 'shining': 2, 'weather': 6, 'sun': 3, 'is': 1, 'sweet': 4,
    'and': 0}~~
{'sun': 4, 'and': 0, 'is': 1, 'the': 6, 'shining': 3, 'two': 7, 'sweet': 5, 'weather': 8, 'one': 2}
```

As we can see from executing the preceding command, the vocabulary is stored in a Python dictionary, which maps the unique words that are mapped to integer indices. Next let us print the feature vectors that we just created:

```
    >>> print(bag.toarray())
    ~~[[0 1 1 1 0 1 0]
     [0 1 0 0 1 1 1]
     [1 2 1 1 1 2 1]]~~
                              [[0 1 0 1 1 0 1 0 0]
                               [0 1 0 0 0 1 1 0 1]
                               [2 3 2 1 1 1 2 1 1]]
```

Each index position in the feature vectors shown here corresponds to the integer values that are stored as dictionary items in the `CountVectorizer` vocabulary. For example, the first feature at index position `0` resembles the count of the word `and`, which only occurs in the last document, and the word `is` at index position `1` (the 2nd feature in the document vectors) occurs in all three sentences. Those values in the feature vectors are also called the **raw term frequencies**: *tf (t,d)*—the number of times a term *t* occurs in a document *d*.

The sequence of items in the bag-of-words model that we just created is also called the **1-gram** or **unigram** model—each item or token in the vocabulary represents a single word. More generally, the contiguous sequences of items in NLP—words, letters, or symbols—is also called an **n-gram**. The choice of the number *n* in the n-gram model depends on the particular application; for example, a study by Kanaris et al. revealed that n-grams of size 3 and 4 yield good performances in anti-spam filtering of e-mail messages (Ioannis Kanaris, Konstantinos Kanaris, Ioannis Houvardas, and Efstathios Stamatatos. *Words vs Character N-Grams for Anti-Spam Filtering*. International Journal on Artificial Intelligence Tools, 16(06):1047–1067, 2007). To summarize the concept of the n-gram representation, the 1-gram and 2-gram representations of our first document "the sun is shining" would be constructed as follows:

- **1-gram**: "the", "sun", "is", "shining"
- **2-gram**: "the sun", "sun is", "is shining"

The `CountVectorizer` class in scikit-learn allows us to use different n-gram models via its `ngram_range` parameter. While a 1-gram representation is used by default, we could switch to a 2-gram representation by initializing a new `CountVectorizer` instance with `ngram_range=(2,2)`.

## Assessing word relevancy via term frequency-inverse document frequency

When we are analyzing text data, we often encounter words that occur across multiple documents from both classes. Those frequently occurring words typically don't contain useful or discriminatory information. In this subsection, we will learn about a useful technique called **term frequency-inverse document frequency** (**tf-idf**) that can be used to downweight those frequently occurring words in the feature vectors. The tf-idf can be defined as the product of the **term frequency** and the **inverse document frequency**:

$$\text{tf-idf}(\text{t,d}) = tf(t,d) \times \text{idf}(\text{t,d})$$

Here the *tf(t, d)* is the term frequency that we introduced in the previous section, and the inverse document frequency *idf(t, d)* can be calculated as:

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

where $n_d$ is the total number of documents, and *df(d, t)* is the number of documents *d* that contain the term *t*. Note that adding the constant 1 to the denominator is optional and serves the purpose of assigning a non-zero value to terms that occur in all training samples; the log is used to ensure that low document frequencies are not given too much weight.

Scikit-learn implements yet another transformer, the `TfidfTransformer`, that takes the raw term frequencies from `CountVectorizer` as input and transforms them into tf-idfs:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf = TfidfTransformer()
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
~~[[ 0.    0.43  0.56  0.56  0.    0.43  0.  ]~~
~~ [ 0.    0.43  0.    0.    0.56  0.43  0.56]~~
~~ [ 0.4   0.48  0.31  0.31  0.31  0.48  0.31]]~~
```

```
[[ 0.    0.43  0.    0.56  0.56  0.    0.43  0.    0.  ]
 [ 0.    0.43  0.    0.    0.    0.56  0.43  0.    0.56]
 [ 0.5   0.45  0.5   0.19  0.19  0.19  0.3   0.25  0.19]]
```

At contrary the word «one» (3rd element of each frequency vector) has a frequency of 2 in the 3rd document but has a greater tf-idf (0.5) since it is only present in the 3rd document, thus more discriminatory.

As we saw in the previous subsection, the word `is` had the largest term frequency in the 3rd document, being the most frequently occurring word. However, after transforming the same feature vector into tf-idfs, we see that the word `is` is now associated with a relatively small tf-idf ~~(0.31)~~ 0.45 in document 3 since it is also contained in documents 1 and 2 and thus is unlikely to contain any useful, discriminatory information.

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

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

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

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

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

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

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

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

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

```
                [3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0 , 1.69, 1.29]
tf-idf_norm =   ---------------------------------------------------------------------------
                sqrt ([3.39**2, 3.0**2, 3.39**2, 1.29**2, 1.29**2, 1.29**2, 2.0**2, 1.69**2, 1.29**2])

            =   [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19]
 tf-idf_norm("is",d3)= 0.45
```

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

$$\text{tf-idf}\left("is", \text{d3}\right) = \overset{3}{\cancel{2}} \times (0+1) = \overset{3}{\cancel{2}}$$

**[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0 , 1.69, 1.29]**

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

$$\text{tf-idf}\left("is", \text{d3}\right) = \overset{0.45}{\cancel{0.48}}$$

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

## Cleaning text data

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A Ridge Regression model can be initialized as follows:

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

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

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

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

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

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

## Turning a linear regression model into a curve – polynomial regression

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

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

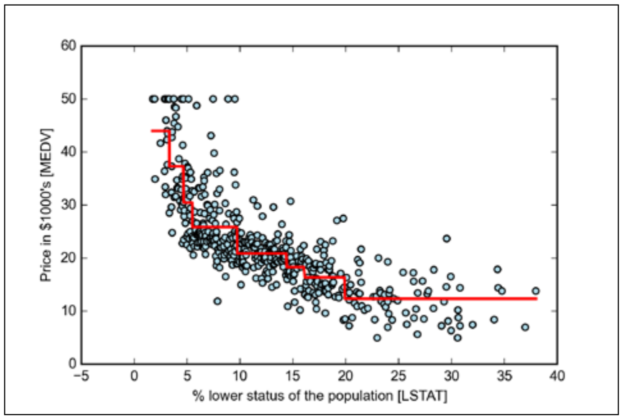
Here, $N_t$ is the number of training samples at node $t$, $D_t$ is the training subset at node $t$, $y^{(i)}$ is the true target value, and $\hat{y}_t$ is the predicted target value (sample mean):

$$\hat{y}_t = \frac{1}{N} \sum_{i \in D_t} y^{(i)}$$

In the context of decision tree regression, the MSE is often also referred to as within-node variance, which is why the splitting criterion is also better known as *variance reduction*. To see what the line fit of a decision tree looks like, let's use the `DecisionTreeRegressor` implemented in scikit-learn to model the nonlinear relationship between the **MEDV** and **LSTAT** variables:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
>>> sort_idx = X.flatten().argsort()
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.show()
```

As we can see from the resulting plot, the decision tree captures the general trend in the data. However, a limitation of this model is that it does not capture the continuity and differentiability of the desired prediction. In addition, we need to be careful about choosing an appropriate value for the depth of the tree to not overfit or underfit the data; here, a depth of 3 seems to be a good choice:



However, note that the membership indicator $w^{(i,j)}$ is not a binary value as in k-means ( $w^{(i,j)} \in \{0,1\}$ ) but a real value that denotes the cluster membership probability ( $w^{(i,j)} \in [0,1]$ ). You also may have noticed that we added an additional exponent to $w^{(i,j)}$; the exponent *m*, any number greater or equal to 1 (typically *m* = 2), is the so-called **fuzziness coefficient** (or simply **fuzzifier**) that controls the degree of **fuzziness**. The larger the value of $\boldsymbol{m}$ , the smaller the cluster membership $w^{(i,j)}$ becomes, which leads to fuzzier clusters. The cluster membership probability itself is calculated as follows:

$$w^{(i,j)} = \left[ \sum_{p=1}^{k} \left( \frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(p)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

For example, if we chose three cluster centers as in the previous k-means example, we could calculate the membership of the $\boldsymbol{x}^{(i)}$ sample belonging to the $\mu^{(j)}$ cluster as:

$$w^{(i,j)} = \left[ \left( \frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left( \frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left( \frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

The center $\mu^{(j)}$ of a cluster itself is calculated as the mean of all samples in the cluster weighted by the membership degree of belonging to its own cluster:

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} w^{m(i,j)} \boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{m(i,j)}}$$

Just by looking at the equation to calculate the cluster memberships, it is intuitive to say that each iteration in FCM is more expensive than an iteration in k-means. However, FCM typically requires fewer iterations overall to reach convergence. Unfortunately, the FCM algorithm is currently not implemented in scikit-learn. However, it has been found in practice that both k-means and FCM produce very similar clustering outputs, as described in a study by Soumi Ghosh and Sanjay K. Dubey (S. Ghosh and S. K. Dubey. *Comparative Analysis of k-means and Fuzzy c-means Algorithms*. IJACSA, 4:35–38, 2013).

The silhouette coefficient is bounded in the range -1 to 1. Based on the preceding formula, we can see that the silhouette coefficient is 0 if the cluster separation and cohesion are equal ($b^{(i)} = a^{(i)}$). Furthermore, we get close to an ideal silhouette coefficient of 1 if $b^{(i)} >> a^{(i)}$, since $b^{(i)}$ quantifies how dissimilar a sample is to other clusters, and $a^{(i)}$ tells us how similar it is to the other samples in its own cluster, respectively.

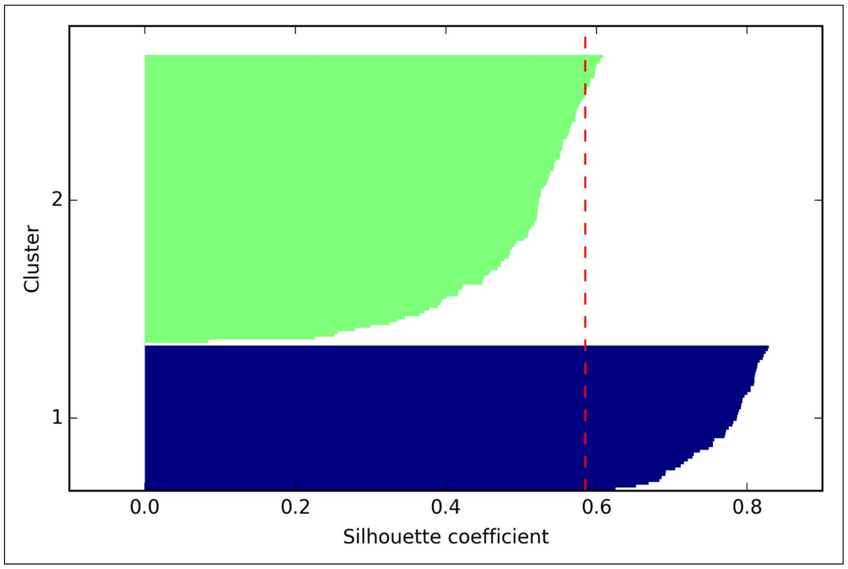
The silhouette coefficient is available as `silhouette_samples` from scikit-learn's `metric` module, and optionally the `silhouette_scores` can be imported. This calculates the average silhouette coefficient across all samples, which is equivalent to `numpy.mean(silhouette_samples(…))`. By executing the following code, we will now create a plot of the silhouette coefficients for a k-means clustering with $k = 3$:

```
>>> km = KMeans(n_clusters=3,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)

>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                      y_km,
...                                      metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)       use float(i) to make it Python 2.7 compatible
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...             color="red",
...             linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
```

```
...                                     y_km,
...                                     metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.show()
```

As we can see in the resulting plot, the silhouettes now have visibly different lengths and width, which yields further evidence for a suboptimal clustering:



In *Chapter 2*, *Training Machine Learning Algorithms for Classification*, we implemented the Adaline algorithm to perform binary classification, and we used a **gradient descent** optimization algorithm to learn the weight coefficients of the model. In every **epoch** (pass over the training set), we updated the weight vector $\boldsymbol{w}$ using the following update rule:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \text{ where } \Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w})$$

In other words, we computed the gradient based on the whole training set and updated the weights of the model by taking a step into the opposite direction of the gradient $\nabla J(\mathbf{w})$. In order to find the optimal weights of the model, we optimized an objective function that we defined as the **Sum of Squared Errors** (**SSE**) cost function $J(\mathbf{w})$. Furthermore, we multiplied the gradient by a factor, the **learning rate** $\eta$, which we chose carefully to balance the speed of learning against the risk of overshooting the global minimum of the cost function.

In gradient descent optimization, we updated all weights simultaneously after each epoch, and we defined the partial derivative for each weight $w_j$ in the weight vector $\boldsymbol{w}$ as follows:

Missing "minus" sign, e.g., see pg. 36

$$\frac{\partial}{\partial w_j} J(\boldsymbol{w}) = -\sum_i \left(y^{(i)} - a^{(i)}\right) x_j^{(i)}$$

Here $y^{(i)}$ is the target class label of a particular sample $x^{(i)}$, and $a^{(i)}$ is the **activation** of the neuron, which is a linear function in the special case of Adaline. Furthermore, we defined the *activation function* $\phi(\cdot)$ as follows:

$$\phi(z) = z = a$$

Here, the net input $z$ is a linear combination of the weights that are connecting the input to the output layer:

$$z = \sum_j w_j x_j = \boldsymbol{w}^T \boldsymbol{x}$$

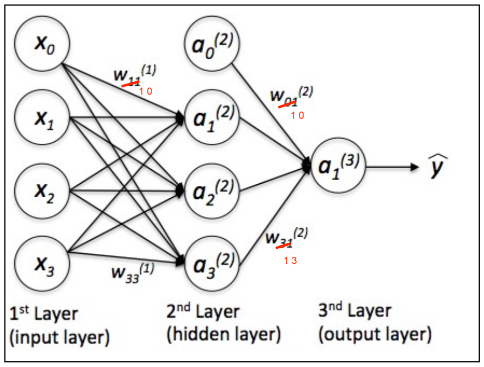
While we used the activation $\phi(z)$ to compute the gradient update, we implemented a **threshold function** (Heaviside function) to squash the continuous-valued output into binary class labels for prediction:

$$\hat{y} = \begin{cases} 1 & if\ g(z) \geq 0 \\ -1 & otherwise \end{cases}$$

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

## Introducing the multi-layer neural network architecture

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



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

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

```
            X_new = np.ones((X.shape[0]+1, X.shape[1]))
            X_new[1:, :] = X
        else:
            raise AttributeError('`how` must be `column` or `row`')
        return X_new

    def _feedforward(self, X, w1, w2):
        a1 = self._add_bias_unit(X, how='column')
        z2 = w1.dot(a1.T)
        a2 = self._sigmoid(z2)
        a2 = self._add_bias_unit(a2, how='row')
        z3 = w2.dot(a2)
        a3 = self._sigmoid(z3)
        return a1, z2, a2, z3, a3

    def _L2_reg(self, lambda_, w1, w2):
        return (lambda_/2.0) * (np.sum(w1[:, 1:] ** 2)\
                + np.sum(w2[:, 1:] ** 2))

    def _L1_reg(self, lambda_, w1, w2):
        return (lambda_/2.0) * (np.abs(w1[:, 1:]).sum()\
                + np.abs(w2[:, 1:]).sum())

    def _get_cost(self, y_enc, output, w1, w2):
        term1 = -y_enc * (np.log(output))
        term2 = (1 - y_enc) * np.log(1 - output)
        cost = np.sum(term1 - term2)
        L1_term = self._L1_reg(self.l1, w1, w2)
        L2_term = self._L2_reg(self.l2, w1, w2)
        cost = cost + L1_term + L2_term
        return cost

    def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):
        # backpropagation
        sigma3 = a3 - y_enc
        z2 = self._add_bias_unit(z2, how='row')
        sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)
        sigma2 = sigma2[1:, :]
        grad1 = sigma2.dot(a1)
        grad2 = sigma3.dot(a2.T)

        # regularize
```

~~grad1[:, 1:] += (w1[:, 1:] * (self.l1 + self.l2))~~

```
      grad1[:, 1:] += self.l2 * w1[:, 1:]
      grad1[:, 1:] += self.l1 * np.sign(w1[:, 1:])
```

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

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

        return grad1, grad2

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

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

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

        for i in range(self.epochs):

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

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

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

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

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

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

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

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

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

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

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

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

$$= a(1-a)$$

To better understand how we compute the $\delta^{(3)}$ term, let's walk through it in more detail. In the preceding equation, we multiplied the transpose $\left(W^{(2)}\right)^T$ of the $t \times h$ dimensional matrix $W^{(2)}$; $t$ is the number of output class labels and $h$ is the number of hidden units. ~~Now, $\left(W^{(2)}\right)^T$ becomes an $h \times t$ dimensional matrix with $\delta^{(3)}$~~, which is a $t \times 1$ dimensional vector. We then performed a pair-wise multiplication between $\left(W^{(2)}\right)^T \delta^{(3)}$ and $\left(a^{(2)} * \left(1-a^{(2)}\right)\right)$, which is also a hx1 ~~$t \times 1$~~ dimensional vector. Eventually, after obtaining the $\delta$ terms, we can now write the derivation of the cost function as follows:

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

**That's a bit confusing, maybe it's more clear to say:**

**"We multiplied (W^(2))^T, an h×t dimensional matrix with Sigma^(3), which is a t x 1 dimensional vector, resulting in a hx1 dimensional vector."**

If we calculate the net input and use it to activate a logistic neuron with those particular feature values and weight coefficients, we get back a value of 0.707, which we can interpret as a 70.7 percent probability that this particular sample $x$ belongs to the positive class. In *Chapter 12*, *Training Artificial Neural Networks for Image Recognition*, we used the one-hot encoding technique to compute the values in the output layer consisting of multiple logistic activation units. However, as we will demonstrate with the following code example, an output layer consisting of multiple logistic activation units does not produce meaningful, interpretable probability values:

```
# W : array, shape = [n_output_units, n_hidden_units+1]
#          Weight matrix for hidden layer -> output layer.
# note that first column ~~(A[:][0] = 1) are the bias units~~
>>> W = np.array([[1.1, 1.2, 1.3, 0.5],  (W[:][0]) contains the bias units
...               [0.1, 0.2, 0.4, 0.1],
...               [0.2, 0.5, 2.1, 1.9]])

# A : array, shape = [n_hidden+1, n_samples]
#          Activation of hidden layer.
# note that first element (A[0][0] = 1) is the bias unit
>>> A = np.array([[1.0],
...               [0.1],
...               [0.3],
...               [0.7]])

# Z : array, shape = [n_output_units, n_samples]
#          Net input of the output layer.
>>> Z = W.dot(A)
>>> y_probas = logistic(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
 [[ 0.87653295]
 [ 0.57688526]
 [ 0.90114393]]
```

As we can see in the output, the probability that the particular sample belongs to the first class is almost 88 percent, the probability that the particular sample belongs to the second class is almost 58 percent, and the probability that the particular sample belongs to the third class is 90 percent, respectively. This is clearly confusing, since we all know that a percentage should intuitively be expressed as a fraction of 100. However, this is in fact not a big concern if we only use our model to predict the class labels, not the class membership probabilities.

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