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:

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

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

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

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

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

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

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

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

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

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

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

\[
\Delta w_j^{(i)} = (1 - 1)0.5 = (2)0.5 = 1
\]

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

\[ w^T x^{(i)} \geq 1 - \xi^{(i)} \text{ if } y^{(i)} = 1 \]

\[ w^T x^{(i)} \leq -1 + \xi^{(i)} \text{ if } y^{(i)} = -1 \]

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

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

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 use the parameter \( C \) to control the width of the margin and therefore tune the bias-variance trade-off as illustrated in the following figure:

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

It should be: “increasing the value of lambda increases the bias ...” (lambda instead of \( C \)) or “decreasing the value of \( C \) increases the bias”
Chapter 3

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:

\[
I_g(t) = \sum_{i=1}^{c} p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2
\]

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

\[
I_g(t) = 1 - \sum_{i=1}^{2} 0.5^2 = 0.5
\]

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:

\[
I_e(t) = 1 - \max \{ p(i|t) \}
\]
Applied to the standardized Wine data, the L1 regularized logistic regression would yield the following sparse solution:

```python
>>> 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:

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

Since we 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:

```python
>>> 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:

\[
z = w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = w^T x
\]
4. Compute the eigenvectors and corresponding eigenvalues of the matrix $S_w S_b$.

5. Choose the $k$ eigenvectors that correspond to the $k$ largest eigenvalues to construct a $d \times k$-dimensional transformation matrix $W$; the eigenvectors are the columns of this matrix.

6. Project the samples onto the new feature subspace using the transformation matrix $W$.

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 $m_i$ stores the mean feature value $\mu_w$ with respect to the samples of class $i$:

$$m_i = \frac{1}{n_i} \sum_{x \in D_i} x$$

This results in three mean vectors:

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

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 for in the later sections.
Compressing Data via Dimensionality Reduction

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:

```python
>>> 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 $S_i$ before we sum them up as scatter matrix $S_w$. 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 $\Sigma$. The covariance matrix is a normalized version of the scatter matrix:

\[
\Sigma = \frac{1}{N_i} S_i = \frac{1}{N_i} \sum_{x \in i} (x - m_i)(x - m_i)^T
\]

```python
>>> 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 $S_B$:

\[
S_B = \sum_{i=1}^{c} N_i (m_i - m)(m_i - m)^T
\]

```python
>>> 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!
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* now associated with a relatively small tf-idf (0.31) in document 3 since it is also contained in documents 1 and 2 and thus is unlikely to contain any useful, discriminatory information.

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:

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

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

\[
\text{tf-idf} (t,d) = \text{tf} (t,d) \times (\text{idf} (t,d) + 1)
\]

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:

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

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

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

\[
idf ("is", d3) = \log \frac{1 + 3}{1 + 3} = 0
\]