

# 7

## Combining Different Models for Ensemble Learning

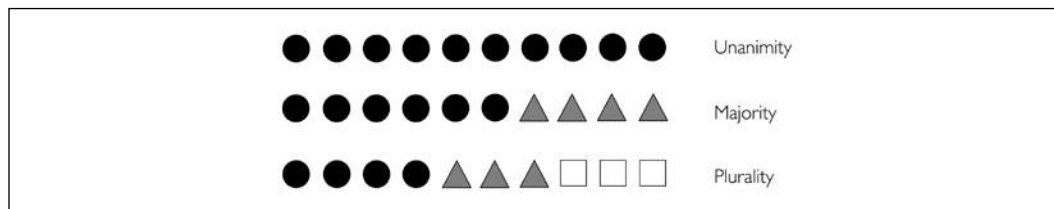
In the previous chapter, we focused on the best practices for tuning and evaluating different models for classification. In this chapter, we will build upon those techniques and explore different methods for constructing a set of classifiers that can often have a better predictive performance than any of its individual members. We will learn how to do the following:

- Make predictions based on majority voting
- Use bagging to reduce overfitting by drawing random combinations of the training dataset with repetition
- Apply boosting to build powerful models from weak learners that learn from their mistakes

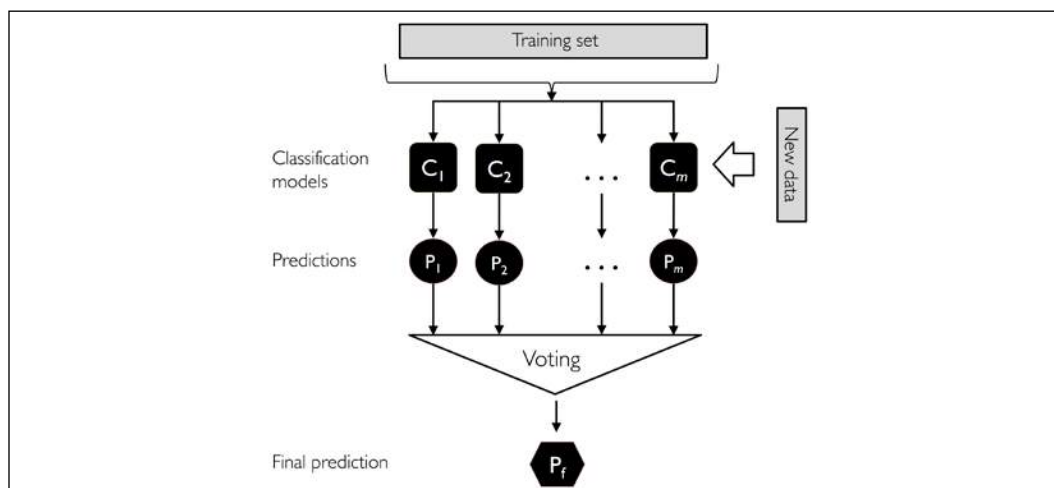
### Learning with ensembles

The goal of **ensemble methods** is to combine different classifiers into a meta-classifier that has better generalization performance than each individual classifier alone. For example, assuming that we collected predictions from 10 experts, ensemble methods would allow us to strategically combine those predictions by the 10 experts to come up with a prediction that was more accurate and robust than the predictions by each individual expert. As you will see later in this chapter, there are several different approaches for creating an ensemble of classifiers. This section will introduce a basic explanation of how ensembles work and why they are typically recognized for yielding a good generalization performance.

In this chapter, we will focus on the most popular ensemble methods that use the **majority voting** principle. Majority voting simply means that we select the class label that has been predicted by the majority of classifiers, that is, received more than 50 percent of the votes. Strictly speaking, the term "majority vote" refers to binary class settings only. However, it is easy to generalize the majority voting principle to multiclass settings, which is called **plurality voting**. Here, we select the class label that received the most votes (the mode). The following diagram illustrates the concept of majority and plurality voting for an ensemble of 10 classifiers, where each unique symbol (triangle, square, and circle) represents a unique class label:



Using the training dataset, we start by training  $m$  different classifiers ( $C_1, \dots, C_m$ ). Depending on the technique, the ensemble can be built from different classification algorithms, for example, decision trees, support vector machines, logistic regression classifiers, and so on. Alternatively, we can also use the same base classification algorithm, fitting different subsets of the training dataset. One prominent example of this approach is the random forest algorithm, which combines different decision tree classifiers. The following figure illustrates the concept of a general ensemble approach using majority voting:



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

$$\hat{y} = \text{mode}\{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}$$

(In statistics, the mode is the most frequent event or result in a set. For example,  $\text{mode}\{1, 2, 1, 1, 2, 4, 5, 4\} = 1$ .)

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

$$C(\mathbf{x}) = \text{sign}\left[\sum_j^m C_j(\mathbf{x})\right] = \begin{cases} 1 & \text{if } \sum_j C_j(\mathbf{x}) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

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 will make the assumption that all  $n$ -base classifiers for a binary classification task have an equal error rate,  $\varepsilon$ . Furthermore, we will 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:

$$P(y \geq k) = \sum_k^n \binom{n}{k} \varepsilon^k (1 - \varepsilon)^{n-k} = \varepsilon_{\text{ensemble}}$$

Here,  $\binom{n}{k}$  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 ( $n = 11$ ), where each classifier has an error rate of 0.25 ( $\varepsilon = 0.25$ ):

$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1 - 0.25)^{11-k} = 0.034$$



### The binomial coefficient

The binomial coefficient refers to the number of ways we can choose subsets of  $k$ -unordered elements from a set of size  $n$ ; thus, it is often called "n choose k." Since the order does not matter here, the binomial coefficient is also sometimes referred to as *combination* or *combinatorial number*, and in its unabbreviated form, it is written as follows:

$$\frac{n!}{(n-k)!k!}$$

Here, the symbol (!) stands for factorial—for example,  $3! = 3 \cdot 2 \cdot 1 = 6$ .

As you 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.special import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = int(math.ceil(n_classifier / 2.))
...     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)
0.03432750701904297
```

After we have 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:

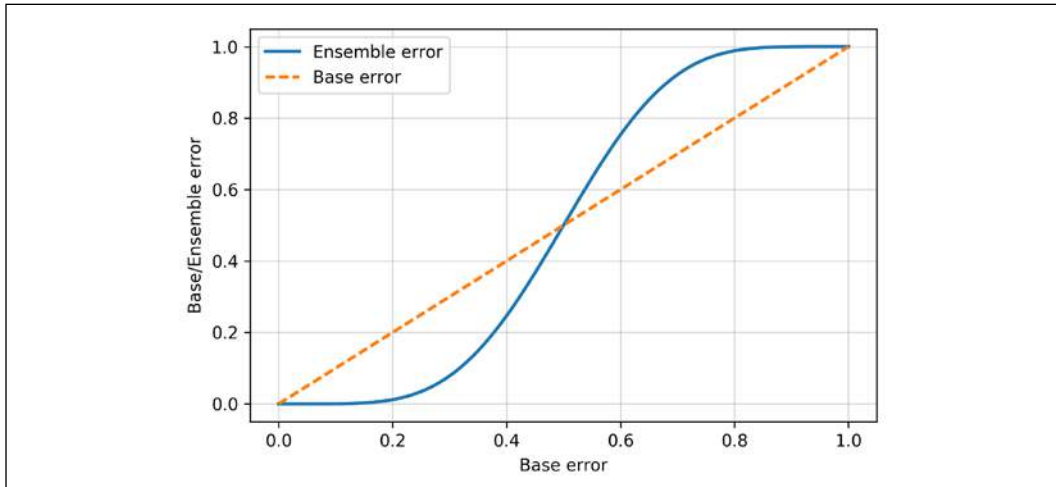
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> plt.plot(error_range, ens_errors,
...          label='Ensemble error',
```

```

...         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(alpha=0.5)
>>> plt.show()

```

As you 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 ( $\epsilon < 0.5$ ). Note that the  $y$  axis depicts the base error (dotted line) as well as the ensemble error (continuous line):



## Combining classifiers via majority vote

After the short introduction to ensemble learning in the previous section, let's start with a warm-up exercise and implement a simple ensemble classifier for majority voting in Python.



### Plurality voting

Although the majority voting algorithm that we will discuss in this section also generalizes to multiclass settings via plurality voting, the term "majority voting" will be used for simplicity, as is often the case in the literature.

## Implementing a simple majority vote classifier

The algorithm that we are going to implement in this section will allow us to combine different classification algorithms associated with individual weights for confidence. Our goal is to build a stronger meta-classifier that balances out the individual classifiers' weaknesses on a particular dataset. In more precise mathematical terms, we can write the weighted majority vote as follows:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(\mathbf{x}) = i)$$

Here,  $w_j$  is a weight associated with a base classifier,  $C_j$ ;  $\hat{y}$  is the predicted class label of the ensemble;  $A$  is the set of unique class labels;  $\chi_A$  (Greek chi) is the characteristic function or indicator function, which returns 1 if the predicted class of the  $j$ th classifier matches  $i$  ( $C_j(\mathbf{x}) = i$ ). For equal weights, we can simplify this equation and write it as follows:

$$\hat{y} = \text{mode}\{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}$$

To better understand the concept of *weighting*, we will now take a look at a more concrete example. Let's assume that we have an ensemble of three base classifiers,  $C_j (j \in \{0, 1\})$ , and we want to predict the class label,  $C_j(\mathbf{x}) \in \{0, 1\}$ , of a given example,  $\mathbf{x}$ . Two out of three base classifiers predict the class label 0, and one,  $C_3$ , predicts that the example belongs to class 1. If we weight the predictions of each base classifier equally, the majority vote predicts that the example belongs to class 0:

$$C_1(\mathbf{x}) \rightarrow 0, \quad C_2(\mathbf{x}) \rightarrow 0, \quad C_3(\mathbf{x}) \rightarrow 1$$

$$\hat{y} = \text{mode}\{0, 0, 1\} = 0$$

Now, let's assign a weight of 0.6 to  $C_3$ , and let's weight  $C_1$  and  $C_2$  by a coefficient of 0.2:

$$\begin{aligned} \hat{y} &= \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(\mathbf{x}) = i) \\ &= \arg \max_i [0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1] = 1 \end{aligned}$$

More simply, since  $3 \times 0.2 = 0.6$ , we can say that the prediction made by  $C_3$  has three times more weight than the predictions by  $C_1$  or  $C_2$ , which we can write as follows:

$$\hat{y} = \text{mode}\{0,0,1,1,1\} = 1$$

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 you will remember from the discussion on logistic regression 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  $j$ th 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 classifiers  $C_j$  return the following class membership probabilities for a particular example,  $\mathbf{x}$ :

$$C_1(\mathbf{x}) \rightarrow [0.9, 0.1], \quad C_2(\mathbf{x}) \rightarrow [0.8, 0.2], \quad C_3(\mathbf{x}) \rightarrow [0.4, 0.6]$$

Using the same weights as previously (0.2, 0.2, and 0.6), we can then calculate the individual class probabilities as follows:

$$\begin{aligned} p(i_0|\mathbf{x}) &= 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58 \\ p(i_1|\mathbf{x}) &= 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42 \\ \hat{y} &= \arg \max_i [p(i_0|\mathbf{x}), p(i_1|\mathbf{x})] = 0 \end{aligned}$$

To implement the weighted majority vote based on class probabilities, we can again make use of NumPy, using `np.average` and `np.argmax`:

```
>>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([0.58, 0.42])
>>> np.argmax(p)
0
```

Putting everything together, let's now implement `MajorityVoteClassifier` in Python:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator

class MajorityVoteClassifier(BaseEstimator,
                           ClassifierMixin):
    """ A majority vote ensemble classifier

    Parameters
    -----
    classifiers : array-like, shape = [n_classifiers]
        Different classifiers for the ensemble

    vote : str, {'classlabel', 'probability'}
        Default: 'classlabel'
        If 'classlabel' the prediction is based on
        the argmax of class labels. Else if
        'probability', the argmax of the sum of
        probabilities is used to predict the class label
        (recommended for calibrated classifiers).

    weights : array-like, shape = [n_classifiers]
        Optional, default: None
        If a list of 'int' or 'float' values are
        provided, the classifiers are weighted by
        importance; Uses uniform weights if 'weights=None'.

    """
    def __init__(self, classifiers,
                 vote='classlabel', weights=None):
```



---

```

self.classifiers = classifiers
self.named_classifiers = {key: value for
                           key, value in
                           _name_estimators(classifiers)}

self.vote = vote
self.weights = weights

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

    Parameters
    -----
    X : {array-like, sparse matrix},
        shape = [n_examples, n_features]
        Matrix of training examples.

    y : array-like, shape = [n_examples]
        Vector of target class labels.

    Returns
    -----
    self : object

    """
    if self.vote not in ('probability', 'classlabel'):
        raise ValueError("vote must be 'probability'"
                          "or 'classlabel'; got (vote=%r)"
                          % self.vote)

    if self.weights and
       len(self.weights) != len(self.classifiers):
        raise ValueError("Number of classifiers and weights"
                          "must be equal; got %d weights,"
                          "%d classifiers"
                          % (len(self.weights),
                             len(self.classifiers)))

    # Use LabelEncoder to ensure class labels start
    # with 0, which is important for np.argmax
    # call in self.predict
    self.lablenc_ = LabelEncoder()
    self.lablenc_.fit(y)
    self.classes_ = self.lablenc_.classes_
    self.classifiers_ = []
    for clf in self.classifiers:
        fitted_clf = clone(clf).fit(X,
                                     self.lablenc_.transform(y))
        self.classifiers_.append(fitted_clf)
    return self

```

I've added a lot of comments to the code to explain the individual parts. However, before we implement the remaining methods, let's take a quick break and discuss some of the code that may look confusing at first. We used the `BaseEstimator` and `ClassifierMixin` parent classes to get some base functionality *for free*, including the `get_params` and `set_params` methods to set and return the classifier's parameters, as well as the `score` method to calculate the prediction accuracy.

Next, we will add the `predict` method to predict the class label via a majority vote based on the class labels if we initialize a new `MajorityVoteClassifier` object with `vote='classlabel'`. Alternatively, we will be able to initialize the ensemble classifier with `vote='probability'` to predict the class label based on the class membership probabilities. Furthermore, we will also add a `predict_proba` method to return the averaged probabilities, which is useful when computing the receiver operating characteristic area under the curve (ROC AUC):

```
def predict(self, X):
    """ Predict class labels for X.

    Parameters
    -----
    X : {array-like, sparse matrix},
        Shape = [n_examples, n_features]
        Matrix of training examples.

    Returns
    -----
    maj_vote : array-like, shape = [n_examples]
        Predicted class labels.

    """
    if self.vote == 'probability':
        maj_vote = np.argmax(self.predict_proba(X), axis=1)
    else: # 'classlabel' vote

        # Collect results from clf.predict calls
        predictions = np.asarray([clf.predict(X)
                                for clf in
                                self.classifiers]).T

        maj_vote = np.apply_along_axis(lambda x: np.argmax(
                                np.bincount(x,
                                weights=self.weights)),
                                axis=1,
                                arr=predictions)
```

---

```

        maj_vote = self.lablenc_.inverse_transform(maj_vote)
        return maj_vote

def predict_proba(self, X):
    """ Predict class probabilities for X.

    Parameters
    -----
    X : {array-like, sparse matrix},
        shape = [n_examples, n_features]
        Training vectors, where
        n_examples is the number of examples and
        n_features is the number of features.

    Returns
    -----
    avg_proba : array-like,
        shape = [n_examples, n_classes]
        Weighted average probability for
        each class per example.

    """
    probas = np.asarray([clf.predict_proba(X)
                        for clf in self.classifiers_])
    avg_proba = np.average(probas, axis=0,
                          weights=self.weights)
    return avg_proba

def get_params(self, deep=True):
    """ Get classifier parameter names for GridSearch"""
    if not deep:
        return super(MajorityVoteClassifier,
                      self).get_params(deep=False)
    else:
        out = self.named_classifiers.copy()
        for name, step in self.named_classifiers.items():
            for key, value in step.get_params(
                deep=True).items():
                out['%s__%s' % (name, key)] = value
        return out

```

Also, note that we defined our own modified version of the `get_params` method to use the `_name_estimators` function to access the parameters of individual classifiers in the ensemble; this may look a little bit complicated at first, but it will make perfect sense when we use grid search for hyperparameter tuning in later sections.



### VotingClassifier in scikit-learn

Although the `MajorityVoteClassifier` implementation is very useful for demonstration purposes, we implemented a more sophisticated version of this majority vote classifier in scikit-learn based on the implementation in the first edition of this book. The ensemble classifier is available as `sklearn.ensemble.VotingClassifier` in scikit-learn version 0.17 and newer.

## Using the majority voting principle to make predictions

Now it is time to put the `MajorityVoteClassifier` that we implemented in the previous section into action. But first, let's prepare a dataset that we can test it on. Since we are already familiar with techniques to load datasets from CSV files, we will take a shortcut and load the Iris dataset from scikit-learn's `datasets` module. Furthermore, we will only select two features, *sepal width* and *petal length*, to make the classification task more challenging for illustration purposes. Although our `MajorityVoteClassifier` generalizes to multiclass problems, we will only classify flower examples from the *Iris-versicolor* and *Iris-virginica* classes, with which we will compute the ROC AUC later. The code is as follows:

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```



### Class membership probabilities from decision trees

Note that scikit-learn uses the `predict_proba` method (if applicable) to compute the ROC AUC score. In *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, we saw how the class probabilities are computed in logistic regression models. In decision trees, the probabilities are calculated from a frequency vector that is created for each node at training time. The vector collects the frequency values of each class label computed from the class label distribution at that node. Then, the frequencies are normalized so that they sum up to 1. Similarly, the class labels of the k-nearest neighbors are aggregated to return the normalized class label frequencies in the k-nearest neighbors algorithm. Although the normalized probabilities returned by both the decision tree and k-nearest neighbors classifier may look similar to the probabilities obtained from a logistic regression model, we have to be aware that these are actually not derived from probability mass functions.

Next, we will split the Iris examples into 50 percent training and 50 percent test data:

```
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                       test_size=0.5,
...                       random_state=1,
...                       stratify=y)
```

Using the training dataset, we now will train three different classifiers:

- Logistic regression classifier
- Decision tree classifier
- k-nearest neighbors classifier

We will then evaluate the model performance of each classifier via 10-fold cross-validation on the training dataset before we combine them into an ensemble classifier:

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
```

```
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           solver='lbfgs',
...                           random_state=1)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])
>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf3]])
>>> clf_labels = ['Logistic regression', 'Decision tree', 'KNN']
>>> print('10-fold cross validation:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')
...     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...           % (scores.mean(), scores.std(), label))
```

The output that we receive, as shown in the following snippet, shows that the predictive performances of the individual classifiers are almost equal:

```
10-fold cross validation:

ROC AUC: 0.92 (+/- 0.15) [Logistic regression]
ROC AUC: 0.87 (+/- 0.18) [Decision tree]
ROC AUC: 0.85 (+/- 0.13) [KNN]
```

You may be wondering why we trained the logistic regression and k-nearest neighbors classifier as part of a pipeline. The reason behind it is that, as discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, both the logistic regression and k-nearest neighbors algorithms (using the Euclidean distance metric) are not scale-invariant, in contrast to decision trees. Although the Iris features are all measured on the same scale (cm), it is a good habit to work with standardized features.

Now, let's move on to the more exciting part and combine the individual classifiers for majority rule voting in our `MajorityVoteClassifier`:

```
>>> mv_clf = MajorityVoteClassifier(
...         classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Majority voting']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')
...     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...           % (scores.mean(), scores.std(), label))
ROC AUC: 0.92 (+/- 0.15) [Logistic regression]
ROC AUC: 0.87 (+/- 0.18) [Decision tree]
ROC AUC: 0.85 (+/- 0.13) [KNN]
ROC AUC: 0.98 (+/- 0.05) [Majority voting]
```

As you can see, the performance of `MajorityVotingClassifier` has improved over the individual classifiers in the 10-fold cross-validation evaluation.

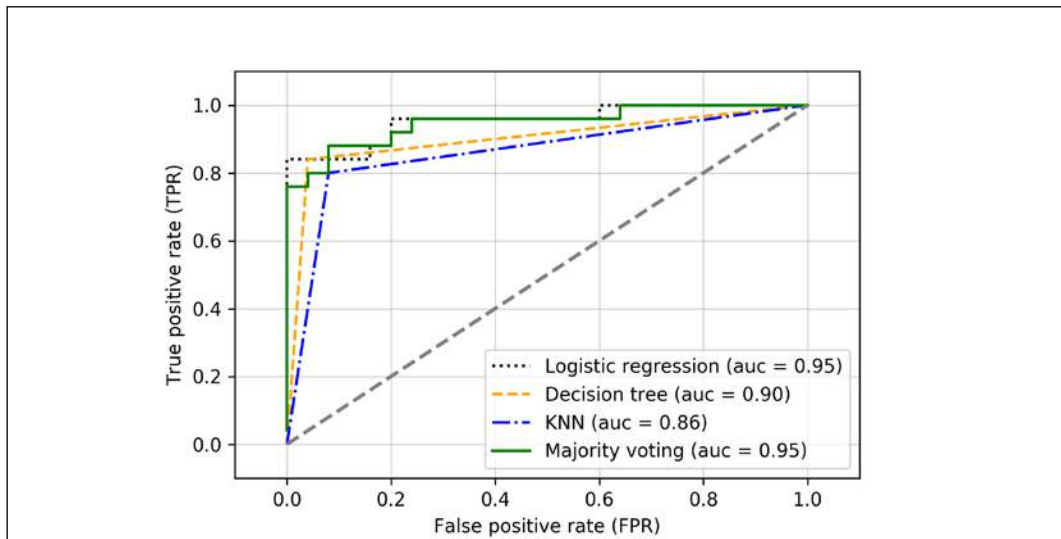
## Evaluating and tuning the ensemble classifier

In this section, we are going to compute the ROC curves from the test dataset to check that `MajorityVoteClassifier` generalizes well with unseen data. We must remember that the test dataset is not to be used for model selection; its purpose is merely to report an unbiased estimate of the generalization performance of a classifier system:

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyle = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...     in zip(all_clf, clf_labels, colors, linestyle):
...     # assuming the label of the positive class is 1
...     y_pred = clf.fit(X_train,
...                     y_train).predict_proba(X_test)[: , 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                       y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...              color=clr,
```

```
...             linestyle=ls,
...             label='%s (auc = %0.2f)' % (label, roc_auc))
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...         linestyle='--',
...         color='gray',
...         linewidth=2)
>>> plt.xlim([-0.1, 1.1])
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid(alpha=0.5)
>>> plt.xlabel('False positive rate (FPR)')
>>> plt.ylabel('True positive rate (TPR)')
>>> plt.show()
```

As you can see in the resulting ROC, the ensemble classifier also performs well on the test dataset (ROC AUC = 0.95). However, you can see that the logistic regression classifier performs similarly well on the same dataset, which is probably due to the high variance (in this case, sensitivity of how we split the dataset) given the small size of the dataset:



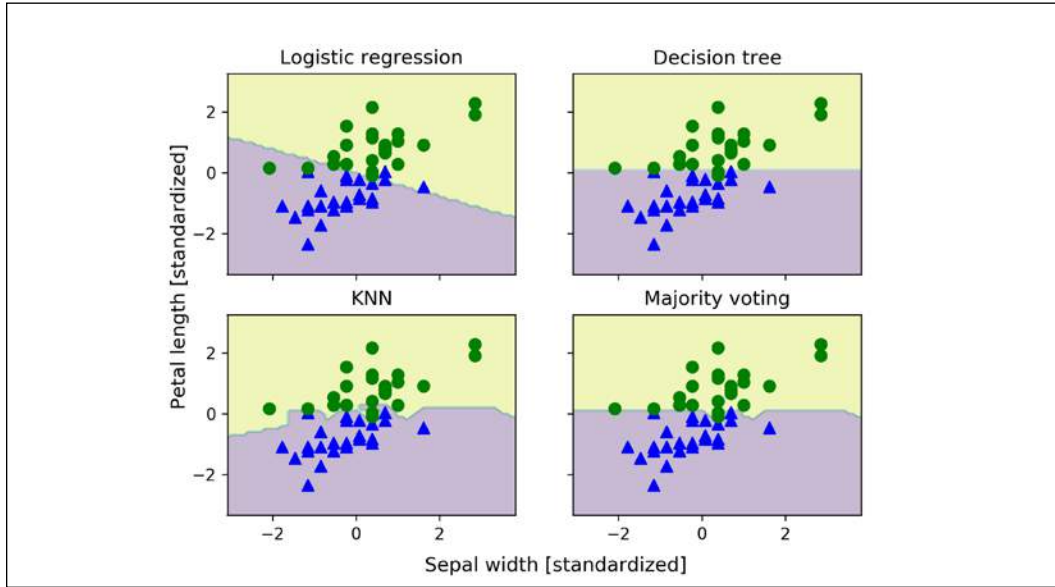
Since we only selected two features for the classification examples, it would be interesting to see what the decision region of the ensemble classifier actually looks like.



Although it is not necessary to standardize the training features prior to model fitting, because our logistic regression and k-nearest neighbors pipelines will automatically take care of it, we will standardize the training dataset so that the decision regions of the decision tree will be on the same scale for visual purposes. The code is as follows:

```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>>
>>> y_max = X_train_std[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                       np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                          sharex='col',
...                          sharey='row',
...                          figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                           all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                     X_train_std[y_train==0, 1],
...                                     c='blue',
...                                     marker='^',
...                                     s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                     X_train_std[y_train==1, 1],
...                                     c='green',
...                                     marker='o',
...                                     s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -5.,
...           s='Sepal width [standardized]',
...           ha='center', va='center', fontsize=12)
>>> plt.text(-12.5, 4.5,
...           s='Petal length [standardized]',
...           ha='center', va='center',
...           fontsize=12, rotation=90)
>>> plt.show()
```

Interestingly, but also as expected, the decision regions of the ensemble classifier seem to be a hybrid of the decision regions from the individual classifiers. At first glance, the majority vote decision boundary looks a lot like the decision of the decision tree stump, which is orthogonal to the  $y$  axis for *sepal width*  $\geq 1$ . However, you can also notice the non-linearity from the k-nearest neighbor classifier mixed in:



Before we tune the individual classifier's parameters for ensemble classification, let's call the `get_params` method to get a basic idea of how we can access the individual parameters inside a `GridSearchCV` object:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier':
  DecisionTreeClassifier(class_weight=None, criterion='entropy',
                        max_depth=1, max_features=None,
                        max_leaf_nodes=None, min_samples_leaf=1,
                        min_samples_split=2,
                        min_weight_fraction_leaf=0.0,
                        random_state=0, splitter='best'),
 'decisiontreeclassifier__class_weight': None,
 'decisiontreeclassifier__criterion': 'entropy',
 [...],
 'decisiontreeclassifier__random_state': 0,
 'decisiontreeclassifier__splitter': 'best',
 'pipeline-1':
```

---

```

Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                     with_std=True)),
                ('clf', LogisticRegression(C=0.001,
                                     class_weight=None,
                                     dual=False,
                                     fit_intercept=True,
                                     intercept_scaling=1,
                                     max_iter=100,
                                     multi_class='ovr',
                                     penalty='l2',
                                     random_state=0,
                                     solver='liblinear',
                                     tol=0.0001,
                                     verbose=0))]),

'pipeline-1__clf':
LogisticRegression(C=0.001, class_weight=None, dual=False,
                  fit_intercept=True, intercept_scaling=1,
                  max_iter=100, multi_class='ovr',
                  penalty='l2', random_state=0,
                  solver='liblinear', tol=0.0001, verbose=0),
'pipeline-1__clf__C': 0.001,
'pipeline-1__clf__class_weight': None,
'pipeline-1__clf__dual': False,
[...]
'pipeline-1__sc__with_std': True,
'pipeline-2':
Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                     with_std=True)),
                ('clf', KNeighborsClassifier(algorithm='auto',
                                     leaf_size=30,
                                     metric='minkowski',
                                     metric_params=None,
                                     n_neighbors=1,
                                     p=2,
                                     weights='uniform'))]),

'pipeline-2__clf':
KNeighborsClassifier(algorithm='auto', leaf_size=30,
                    metric='minkowski', metric_params=None,
                    n_neighbors=1, p=2, weights='uniform'),
'pipeline-2__clf__algorithm': 'auto',
[...]
'pipeline-2__sc__with_std': True}

```

Based on the values returned by the `get_params` method, we now know how to access the individual classifier's attributes. Let's now tune the inverse regularization parameter, `C`, of the logistic regression classifier and the decision tree depth via a grid search for demonstration purposes:

```
>>> from sklearn.model_selection import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...           'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                      param_grid=params,
...                      cv=10,
...                      iid=False,
...                      scoring='roc_auc')
>>> grid.fit(X_train, y_train)
```

After the grid search has completed, we can print the different hyperparameter value combinations and the average ROC AUC scores computed via 10-fold cross-validation as follows:

```
>>> for r, _ in enumerate(grid.cv_results_['mean_test_score']):
...     print("%0.3f +/- %0.2f %r"
...           % (grid.cv_results_['mean_test_score'][r],
...              grid.cv_results_['std_test_score'][r] / 2.0,
...              grid.cv_results_['params'][r]))
0.944 +/- 0.07 {'decisiontreeclassifier__max_depth': 1,
               'pipeline-1__clf__C': 0.001}
0.956 +/- 0.07 {'decisiontreeclassifier__max_depth': 1,
               'pipeline-1__clf__C': 0.1}
0.978 +/- 0.03 {'decisiontreeclassifier__max_depth': 1,
               'pipeline-1__clf__C': 100.0}
0.956 +/- 0.07 {'decisiontreeclassifier__max_depth': 2,
               'pipeline-1__clf__C': 0.001}
0.956 +/- 0.07 {'decisiontreeclassifier__max_depth': 2,
               'pipeline-1__clf__C': 0.1}
0.978 +/- 0.03 {'decisiontreeclassifier__max_depth': 2,
               'pipeline-1__clf__C': 100.0}

>>> print('Best parameters: %s' % grid.best_params_)
Best parameters: {'decisiontreeclassifier__max_depth': 1,
                  'pipeline-1__clf__C': 0.001}

>>> print('Accuracy: %.2f' % grid.best_score_)
Accuracy: 0.98
```

As you can see, we get the best cross-validation results when we choose a lower regularization strength ( $C=0.001$ ), whereas the tree depth does not seem to affect the performance at all, suggesting that a decision stump is sufficient to separate the data. To remind ourselves that it is a bad practice to use the test dataset more than once for model evaluation, we are not going to estimate the generalization performance of the tuned hyperparameters in this section. We will move on swiftly to an alternative approach for ensemble learning: **bagging**.



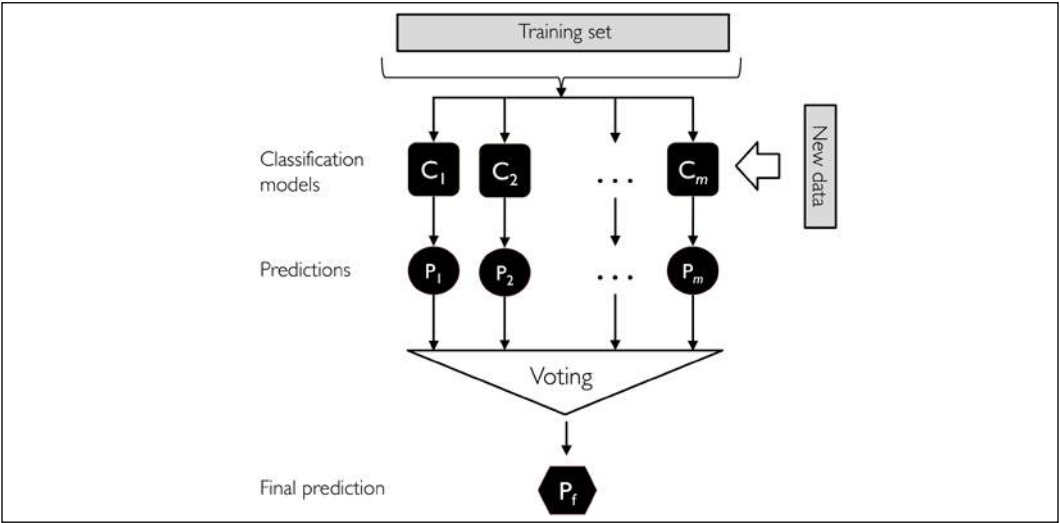
### Building ensembles using stacking

The majority vote approach we implemented in this section is not to be confused with **stacking**. The stacking algorithm can be understood as a two-level ensemble, where the first level consists of individual classifiers that feed their predictions to the second level, where another classifier (typically logistic regression) is fit to the level-one classifier predictions to make the final predictions. The stacking algorithm has been described in more detail by David H. Wolpert in *Stacked generalization, Neural Networks*, 5(2):241–259, 1992. Unfortunately, an implementation of this algorithm has not been implemented in scikit-learn at the time of writing; however, this feature is underway. In the meantime, you can find scikit-learn-compatible implementations of stacking at [http://rasbt.github.io/mlxtend/user\\_guide/classifier/StackingClassifier/](http://rasbt.github.io/mlxtend/user_guide/classifier/StackingClassifier/) and [http://rasbt.github.io/mlxtend/user\\_guide/classifier/StackingCVClassifier/](http://rasbt.github.io/mlxtend/user_guide/classifier/StackingCVClassifier/).

## Bagging – building an ensemble of classifiers from bootstrap samples

Bagging is an ensemble learning technique that is closely related to the `MajorityVoteClassifier` that we implemented in the previous section. However, instead of using the same training dataset to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training dataset, which is why bagging is also known as *bootstrap aggregating*.

The concept of bagging is summarized in the following diagram:



In the following subsections, we will work through a simple example of bagging by hand and use scikit-learn for classifying wine examples.

## Bagging in a nutshell

To provide a more concrete example of how the bootstrap aggregating of a bagging classifier works, let's consider the example shown in the following figure. Here, we have seven different training instances (denoted as indices 1-7) that are sampled randomly with replacement in each round of bagging. Each bootstrap sample is then used to fit a classifier,  $C_j$ , which is most typically an unpruned decision tree:

Sample indices	Bagging round 1	Bagging round 2	...
1	2	7	...
2	2	3	...
3	1	2	...
4	3	1	...
5	7	1	...
6	2	7	...
7	4	7	...
	$C_1$	$C_2$	$C_m$

As you can see from the previous illustration, each classifier receives a random subset of examples from the training dataset. We denote these random samples obtained via bagging as *Bagging round 1*, *Bagging round 2*, and so on. Each subset contains a certain portion of duplicates and some of the original examples don't appear in a resampled dataset at all due to sampling with replacement. Once the individual classifiers are fit to the bootstrap samples, the predictions are combined using majority voting.

Note that bagging is also related to the random forest classifier that we introduced in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*. In fact, random forests are a special case of bagging where we also use random feature subsets when fitting the individual decision trees.



### Model ensembles using bagging

Bagging was first proposed by Leo Breiman in a technical report in 1994; he also showed that bagging can improve the accuracy of unstable models and decrease the degree of overfitting. I highly recommend that you read about his research in *Bagging predictors*, L. Breiman, *Machine Learning*, 24(2):123–140, 1996, which is freely available online, to learn more details about bagging.

## Applying bagging to classify examples in the Wine dataset

To see bagging in action, let's create a more complex classification problem using the Wine dataset that was introduced in *Chapter 4, Building Good Training Datasets – Data Preprocessing*. Here, we will only consider the Wine classes 2 and 3, and we will select two features – Alcohol and OD280/OD315 of diluted wines:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                       'machine-learning-databases/wine/wine.data',
...                       header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash',
...                    'Magnesium', 'Total phenols',
...                    'Flavanoids', 'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
```

```
...                'Proline']
>>> # drop 1 class
>>> df_wine = df_wine[df_wine['Class label'] != 1]
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol',
...             'OD280/OD315 of diluted wines']].values
```

Next, we will encode the class labels into binary format and split the dataset into 80 percent training and 20 percent test datasets, respectively:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.model_selection import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                       test_size=0.2,
...                       random_state=1,
...                       stratify=y)
```



### Obtaining the Wine dataset

You can find a copy of the Wine dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the UCI server at <https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data> is temporarily unavailable. For instance, to load the Wine dataset from a local directory, take the following lines:

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

and replace them with these:

```
df = pd.read_csv(
    'your/local/path/to/wine.data',
    header=None)
```



A `BaggingClassifier` algorithm is already implemented in `scikit-learn`, which we can import from the `ensemble` submodule. Here, we will use an unpruned decision tree as the base classifier and create an ensemble of 500 decision trees fit on different bootstrap samples of the training dataset:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               random_state=1,
...                               max_depth=None)
>>> bag = BaggingClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          max_samples=1.0,
...                          max_features=1.0,
...                          bootstrap=True,
...                          bootstrap_features=False,
...                          n_jobs=1,
...                          random_state=1)
```

Next, we will calculate the accuracy score of the prediction on the training and test datasets to compare the performance of the bagging classifier to the performance of a single unpruned decision tree:

```
>>> from sklearn.metrics import accuracy_score
>>> 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/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 1.000/0.833
```

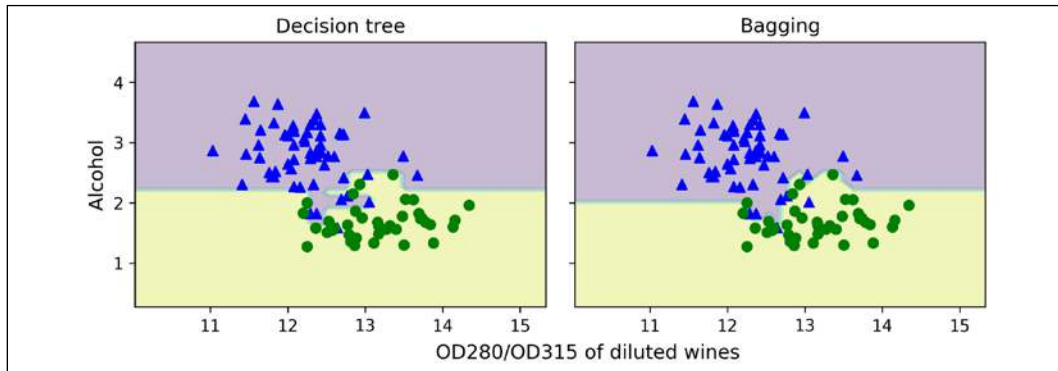
Based on the accuracy values that we printed here, the unpruned decision tree predicts all the class labels of the training examples correctly; however, the substantially lower test accuracy indicates high variance (overfitting) of the model:

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Bagging train/test accuracies %.3f/%.3f'
...       % (bag_train, bag_test))
Bagging train/test accuracies 1.000/0.917
```

Although the training accuracies of the decision tree and bagging classifier are similar on the training dataset (both 100 percent), we can see that the bagging classifier has a slightly better generalization performance, as estimated on the test dataset. Next, let's compare the decision regions between the decision tree and the bagging classifier:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                       np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                          sharex='col',
...                          sharey='row',
...                          figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                          [tree, bag],
...                          ['Decision tree', 'Bagging']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                         X_train[y_train==0, 1],
...                         c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                         X_train[y_train==1, 1],
...                         c='green', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...          s='OD280/OD315 of diluted wines',
...          ha='center',
...          va='center',
...          fontsize=12,
...          transform=axarr[1].transAxes)
>>> plt.show()
```

As we can see in the resulting plot, the piece-wise linear decision boundary of the three-node deep decision tree looks smoother in the bagging ensemble:



We only looked at a very simple bagging example in this section. In practice, more complex classification tasks and a dataset's high dimensionality can easily lead to overfitting in single decision trees, and this is where the bagging algorithm can really play to its strengths. Finally, we must note that the bagging algorithm can be an effective approach to reducing the variance of a model. However, bagging is ineffective in reducing model bias, that is, models that are too simple to capture the trend in the data well. This is why we want to perform bagging on an ensemble of classifiers with low bias, for example, unpruned decision trees.

## Leveraging weak learners via adaptive boosting

In this last section about ensemble methods, we will discuss **boosting**, with a special focus on its most common implementation: **Adaptive Boosting (AdaBoost)**.



### AdaBoost recognition

The original idea behind AdaBoost was formulated by Robert E. Schapire in 1990. *The Strength of Weak Learnability*, R. E. Schapire, *Machine Learning*, 5(2): 197-227, 1990. After Robert Schapire and Yoav Freund presented the AdaBoost algorithm in the *Proceedings of the Thirteenth International Conference (ICML 1996)*, AdaBoost became one of the most widely used ensemble methods in the years that followed (*Experiments with a New Boosting Algorithm* by Y. Freund, R. E. Schapire, and others, *ICML*, volume 96, 148-156, 1996). In 2003, Freund and Schapire received the Goedel Prize for their groundbreaking work, which is a prestigious prize for the most outstanding publications in the field of computer science.

In boosting, the ensemble consists of very simple base classifiers, also often referred to as **weak learners**, which often only have a slight performance advantage over random guessing—a typical example of a weak learner is a decision tree stump. The key concept behind boosting is to focus on training examples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training examples to improve the performance of the ensemble.

The following subsections will introduce the algorithmic procedure behind the general concept of boosting and AdaBoost. Lastly, we will use scikit-learn for a practical classification example.

## How boosting works

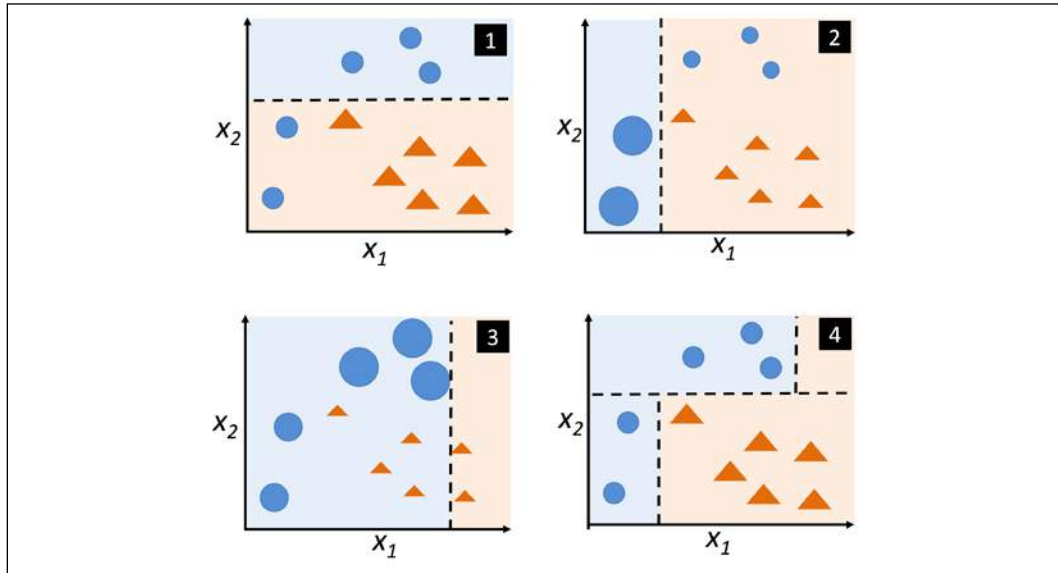
In contrast to bagging, the initial formulation of the boosting algorithm uses random subsets of training examples drawn from the training dataset without replacement; the original boosting procedure can be summarized in the following four key steps:

1. Draw a random subset (sample) of training examples,  $d_1$ , without replacement from the training dataset,  $D$ , to train a weak learner,  $C_1$ .
2. Draw a second random training subset,  $d_2$ , without replacement from the training dataset and add 50 percent of the examples that were previously misclassified to train a weak learner,  $C_2$ .
3. Find the training examples,  $d_3$ , in the training dataset,  $D$ , which  $C_1$  and  $C_2$  disagree upon, to train a third weak learner,  $C_3$ .
4. Combine the weak learners  $C_1$ ,  $C_2$ , and  $C_3$  via majority voting.

As discussed by Leo Breiman (*Bias, variance, and arcing classifiers*, L. Breiman, 1996), boosting can lead to a decrease in bias as well as variance compared to bagging models. In practice, however, boosting algorithms such as AdaBoost are also known for their high variance, that is, the tendency to overfit the training data (*An improvement of AdaBoost to avoid overfitting*, G. Raetsch, T. Onoda, and K. R. Mueller. *Proceedings of the International Conference on Neural Information Processing, CiteSeer*, 1998).

In contrast to the original boosting procedure described here, AdaBoost uses the complete training dataset to train the weak learners, where the training examples are reweighted in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble.

Before we dive deeper into the specific details of the AdaBoost algorithm, let's take a look at the following figure to get a better grasp of the basic concept behind AdaBoost:



To walk through the AdaBoost illustration step by step, we will start with subfigure **1**, which represents a training dataset for binary classification where all training examples are assigned equal weights. Based on this training dataset, we train a decision stump (shown as a dashed line) that tries to classify the examples of the two classes (triangles and circles), as well as possibly 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 examples (circles). Furthermore, we lower the weight of the correctly classified examples. The next decision stump will now be more focused on the training examples that have the largest weights – the training examples that are supposedly hard to classify. The weak learner shown in subfigure **2** misclassifies three different examples 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 then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote, as shown in subfigure **4**.

Now that we have a better understanding of 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 ( $\cdot$ ):

1. Set the weight vector,  $\mathbf{w}$ , to uniform weights, where  $\sum_i w_i = 1$ .
2. For  $j$  in  $m$  boosting rounds, do the following:
  - a. Train a weighted weak learner:  $C_j = \text{train}(\mathbf{X}, \mathbf{y}, \mathbf{w})$ .
  - b. Predict class labels:  $\hat{\mathbf{y}} = \text{predict}(C_j, \mathbf{X})$ .
  - c. Compute weighted error rate:  $\varepsilon = \mathbf{w} \cdot (\hat{\mathbf{y}} \neq \mathbf{y})$ .
  - d. Compute coefficient:  $\alpha_j = 0.5 \log \frac{1 - \varepsilon}{\varepsilon}$
  - e. Update weights:  $\mathbf{w} := \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$ .
  - f. Normalize weights to sum to 1:  $\mathbf{w} := \mathbf{w} / \sum_i w_i$ .
3. Compute the final prediction:  $\hat{\mathbf{y}} = \left( \sum_{j=1}^m (\alpha_j \times \text{predict}(C_j, \mathbf{X})) > 0 \right)$ .

Note that the expression  $(\hat{\mathbf{y}} \neq \mathbf{y})$  in step 2c refers to a binary vector consisting of 1s and 0s, where a 1 is assigned if the prediction is incorrect and 0 is assigned otherwise.

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

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

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

Since the computation of the weight updates may look a little bit complicated at first, we will now follow the calculation step by step. We will start by computing the weighted error rate,  $\varepsilon$ , as described in step 2c:

$$\begin{aligned}\varepsilon &= 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 \\ &\quad + 0.1 \times 1 + 0.1 \times 1 + 0.1 \times 1 + 0.1 \times 0 = \frac{3}{10} = 0.3\end{aligned}$$

Next, we will compute the coefficient,  $\alpha_j$ —shown in step 2d—which will later be used in step 2e to update the weights, as well as for the weights in the majority vote prediction (step 3):

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

After we have computed the coefficient,  $\alpha_j$ , we can now update the weight vector using the following equation:

$$\mathbf{w} := \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$$

Here,  $\hat{\mathbf{y}} \times \mathbf{y}$  is an element-wise multiplication between the vectors of the predicted and true class labels, respectively. Thus, if a prediction,  $\hat{y}_i$ , is correct,  $\hat{\mathbf{y}}_i \times \mathbf{y}_i$  will have a positive sign so that we decrease the  $i$ th weight, since  $\alpha_j$  is a positive number as well:

$$0.1 \times \exp(-0.424 \times 1 \times 1) \approx 0.065$$

Similarly, we will increase the  $i$ th weight if  $\hat{y}_i$  predicted the label incorrectly, like this:

$$0.1 \times \exp(-0.424 \times 1 \times (-1)) \approx 0.153$$

Alternatively, it's like this:

$$0.1 \times \exp(-0.424 \times (-1) \times (1)) \approx 0.153$$

After we have updated each weight in the weight vector, we normalize the weights so that they sum up to 1 (step 2f):

$$\mathbf{w} := \frac{\mathbf{w}}{\sum_i w_i}$$

Here,  $\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$ .

Thus, each weight that corresponds to a correctly classified example will be reduced from the initial value of 0.1 to  $0.065/0.914 \approx 0.071$  for the next round of boosting. Similarly, the weights of the incorrectly classified examples will increase from 0.1 to  $0.153/0.914 \approx 0.167$ .

## Applying AdaBoost using scikit-learn

The previous subsection introduced 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',
...                               random_state=1,
...                               max_depth=1)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                           n_estimators=500,
...                           learning_rate=0.1,
...                           random_state=1)
>>> 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/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.916/0.875
```

As you can see, the decision tree stump seems to underfit the training data in contrast to 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 1.000/0.917
```

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

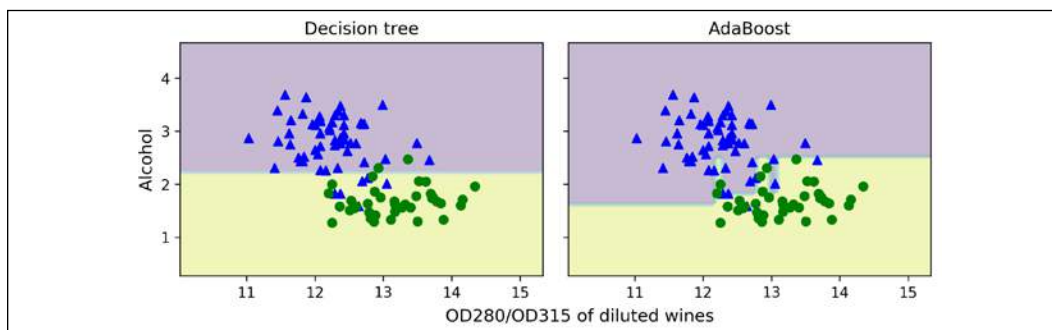
Although we used another simple example for demonstration purposes, we can see that the performance of the AdaBoost classifier is slightly improved compared to the decision stump and achieved very similar accuracy scores as the bagging classifier that we trained in the previous section. However, we must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

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

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

```
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...         s='OD280/OD315 of diluted wines',
...         ha='center',
...         va='center',
...         fontsize=12,
...         transform=axarr[1].transAxes)
>>> plt.show()
```

By looking at the decision regions, you can see that the decision boundary of the AdaBoost model is substantially more complex than the decision boundary of the decision stump. In addition, note that the AdaBoost model separates the feature space very similarly to the bagging classifier that we trained in the previous section:



As concluding remarks about ensemble techniques, it is worth noting that ensemble learning increases the computational complexity compared to individual classifiers. In practice, we need to think carefully about whether we want to pay the price of increased computational costs for an often relatively modest improvement in predictive performance.

An often-cited example of this tradeoff is the famous \$1 million *Netflix Prize*, which was won using ensemble techniques. The details about the algorithm were published in *The BigChaos Solution to the Netflix Grand Prize* by A. Toescher, M. Jahrer, and R. M. Bell, *Netflix Prize documentation, 2009*, which is available at [http://www.stat.osu.edu/~dmsl/GrandPrize2009\\_BPC\\_BigChaos.pdf](http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf). The winning team received the \$1 million grand prize money; however, Netflix never implemented their model due to its complexity, which made it infeasible for a real-world application:

*"We evaluated some of the new methods offline but the additional accuracy gains that we measured did not seem to justify the engineering effort needed to bring them into a production environment."*

<http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html>



## Gradient boosting

Another popular variant of boosting is **gradient boosting**. AdaBoost and gradient boosting share the main overall concept: boosting weak learners (such as decision tree stumps) to strong learners. The two approaches, adaptive and gradient boosting, differ mainly with regard to how the weights are updated and how the (weak) classifiers are combined. If you are familiar with gradient-based optimization and interested in gradient boosting, I recommend reading Jerome Friedman's work (*Greedy function approximation: a gradient boosting machine*. Jerome Friedman. *Annals of Statistics* 2001, pp.1189-1232) and the more recent paper on XGBoost, which is essentially a computationally efficient implementation of the original gradient boost algorithm (*XGBoost: A scalable tree boosting system*. Tianqi Chen and Carlos Guestrin. *Proceeding of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM 2016, pp. 785-794). Note that next to the `GradientBoostingClassifier` implementation in scikit-learn, scikit-learn now also includes a substantially faster version of gradient boosting in version 0.21, `HistGradientBoostingClassifier`, which is even faster than XGBoost. For more information about the `GradientBoostingClassifier` and `HistGradientBoostingClassifier` in scikit-learn, you can read the documentation at <https://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting>. Also, a short, general explanation of gradient boosting can be found in the lecture notes at [https://sebastianraschka.com/pdf/lecture-notes/stat479fs19/07-ensembles\\_\\_notes.pdf](https://sebastianraschka.com/pdf/lecture-notes/stat479fs19/07-ensembles__notes.pdf)

## Summary

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

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

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