# 7_Combining_Different_Models_For_Ensemble_Learning

June 29, 2018

## 1 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 these 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 set with repetition
- Apply boosting to build powerful models from *weak learners* that learn from their mistakes

## 2 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 these predictions by the 10 experts to come up with a prediction that is more accurate and robust than the predictions bt each individual expert. As we will see later in this chapter, there are several different approaches for creating an ensemble of classifiers. In this section, we will introduce a basic perception 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 method 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 multi-class settings, which is called **plurality voting**. Here, we select the class label that received the most votes (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 set, we start by training $m$ different classifiers ($C_1, \ldots, C_m$). Depending on the technique, the ensemble can be built from different classification algorithms, for example, decision trees, suport vector machines, logistic regression classifiers, and so on. Alternatively, we can also use the same class classification algorithm, fitting different subsets of the training set. 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 combine the predicted class labels of each individual classifier and select the class label that received the most votes.

To illustrate why ensemble methods can work betten 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 $\epsilon$. 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:

$$P(y \geq k) = \sum_{k}^{n} \binom{n}{k} \epsilon^k (1 - 0.25)^{n-k} = \epsilon_{ensemble}$$

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 ($\epsilon = 0.25$):

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

As we can see, the error rate of 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 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:

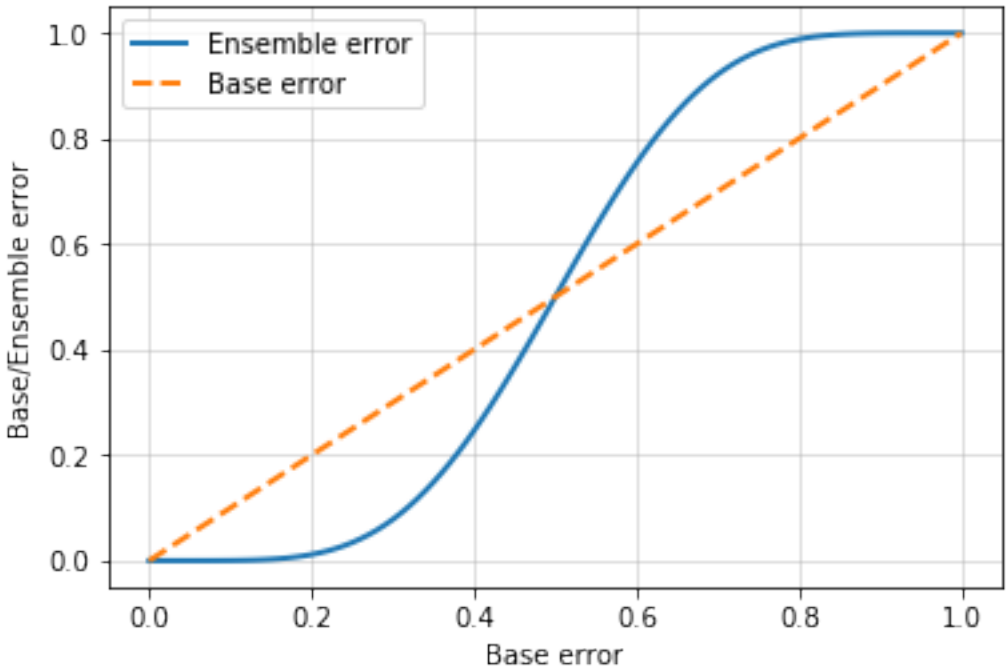
```
In [15]: from scipy.special import comb
         import math

         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
             return sum(probs)

         ensemble_error(n_classifier=11, error=0.25)
```

```
Out[15]: 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:

```
In [16]: import numpy as np
         import matplotlib.pyplot as plt
         %matplotlib inline

         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 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 betten 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).

## 2.1 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.

Although the majority voting algorithm that we will discuss in this section also generalizes to multi-class settings via plurarity voting, we will use the term majority voting for simplicity, as it is also often done in the literature.

## 2.2 Implementing a simple majority vote classifier

The algorithm we we are going ot 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's weaknesses on a particular dataset.

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

```
In [17]: import numpy as np

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

```
Out[17]: 1
```

As we remember from the discussion on logistic regression, 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.

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

```
In [18]: 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
```

```
Out[18]: array([0.58, 0.42])
```

Putting everything together, let's new implement *MajorityVoteClassifier* in Python:

```
In [19]: from sklearn.base import BaseEstimator
         from sklearn.base import ClassifierMixin
         from sklearn.preprocessing import LabelEncoder
         from sklearn.externals import six
         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 the
        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_samples, n_features]
            Matrix of training samples.
        y : array-like, shape = [n_samples]
            Vector of target class labels.

        Returns
        --------------
        self : object
        """

        # 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

    def predict(self, X):
```

```
        """ Predict class labels for X.

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

        Returns
        --------------
        maj_vote : array-like, shape = [n_samples]
            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_samples, n_features]
            Training vectors, where n_samples is
            the number of samples and n_features is the
            number of features.

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

        probas = np.array([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 six.iteritems(self.named_classifiers):
                    for key, value in six.iteritems(step.get_params(deep=True)):
                        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 paramters 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.

### 2.3 Using the majority voting principle to make predictions

Now it is about time to put the *MajorityVoteClassifier* that we implemented 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 dataset 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 samples from the *Iris-versicolor* and *Iris-virginica* classes, with which we will compute the ROC AUC later. The code is as follows:

```
In [20]: 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)
```

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

```
In [21]: 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 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:

```
In [22]: 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, 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))
```

```
10-fold cross validation:

ROC AUC: 0.87 (+/- 0.17) [Logistic regression]
ROC AUC: 0.89 (+/- 0.16) [Decision tree]
ROC AUC: 0.88 (+/- 0.15) [KNN]
```

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

```
In [23]: 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('Accuracy: %0.2f (+/- %0.2f) [%s]'
                   % (scores.mean(), scores.std(), label))
```

```
Accuracy: 0.87 (+/- 0.17) [Logistic regression]
Accuracy: 0.89 (+/- 0.16) [Decision tree]
Accuracy: 0.88 (+/- 0.15) [KNN]
```

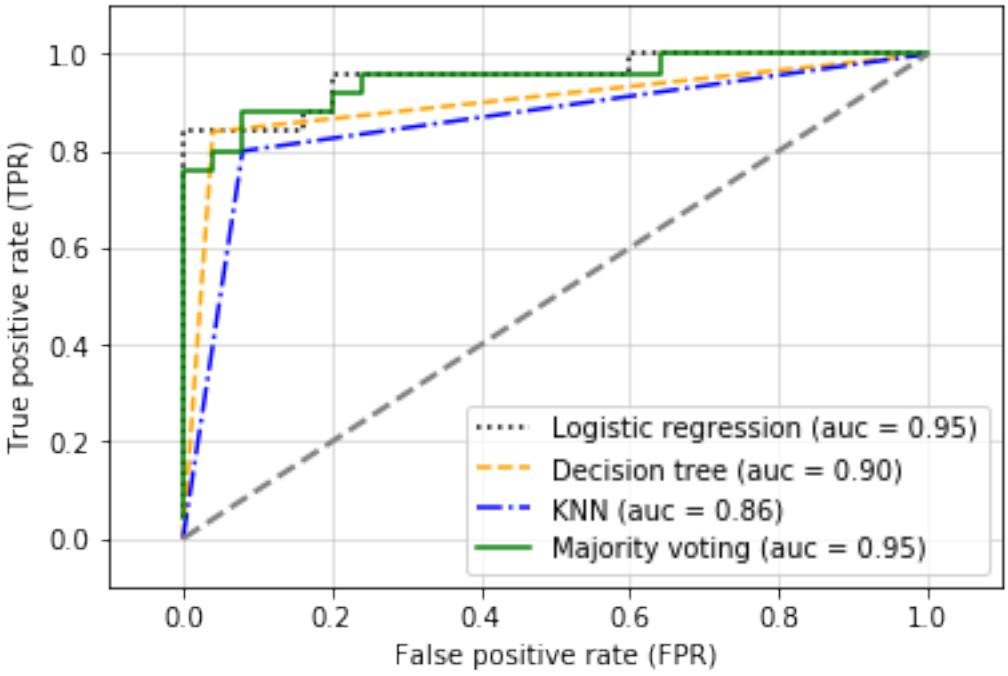
```
Accuracy: 0.94 (+/- 0.13) [Majority voting]
```

As we can see, the performance of *MajorityVoteClassifier* has improved over the individual classifiers in the 10-fold cross-validation evaluation.

### 2.4 Evaluating and tuning the ensemble classifier

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

```
In [24]: from sklearn.metrics import roc_curve
         from sklearn.metrics import auc

         colors = ['black', 'orange', 'blue', 'green']
         linestyles = [':', '--', '-.', '-']
         for clf, label, clr, ls in zip(all_clf, clf_labels, colors, linestyles):
             # 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()
```



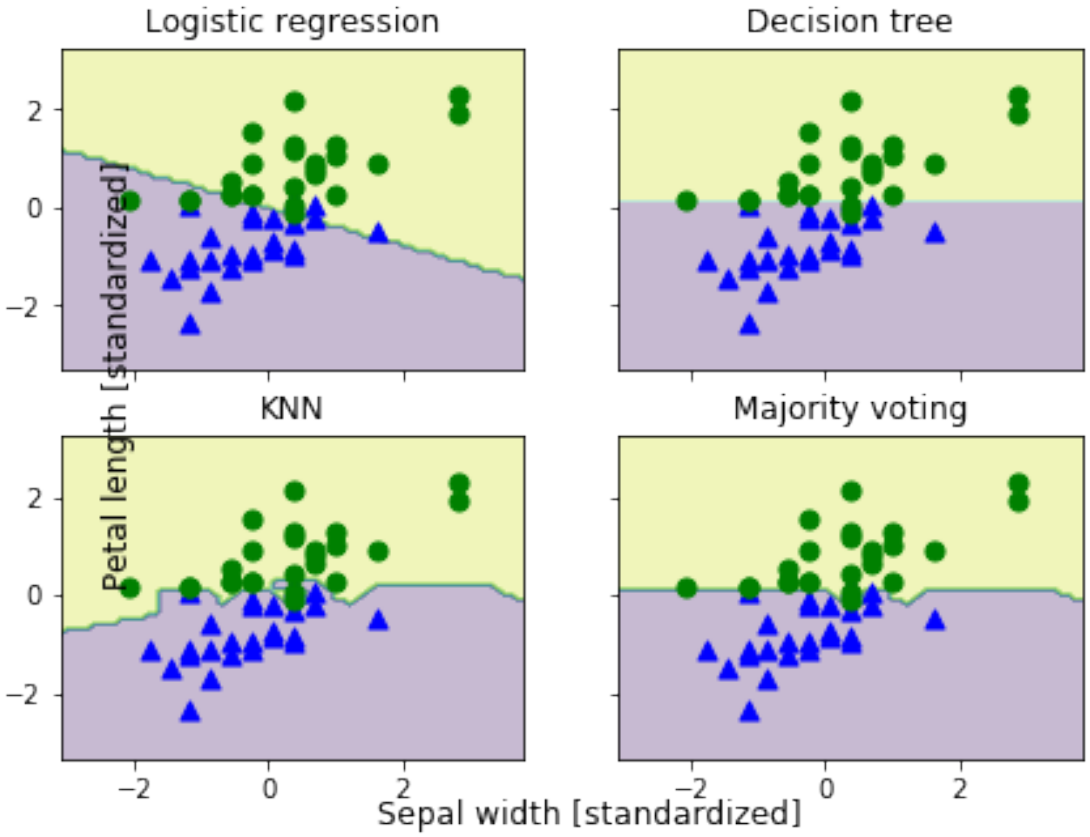

As we can see in the resulting ROC, the ensemble classifier also performs well on the test set (ROC AUC = 0.95). However, we can see that the logistic regression classifier performs similarly well on the same dataset, which is probably due to the high variance (in this case, sensitivity of how we split 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 set so that the decision regions of the decision tree will be on the same scale for visual purposes. The code is as follows:

```
In [25]: from itertools import product

         sc = StandardScaler()
         X_train_std = sc.fit_transform(X_train)

         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, -4.5, s='Sepal width [standardized]',
         ha='center', va='center', fontsize=12)
plt.text(-10.5, 4.5, s='Petal length [standardized]',
         ha='center', va='center', fontsize=12, rotation=90)
plt.show()
```

```
/usr/lib/python3.6/site-packages/sklearn/preprocessing/label.py:151: DeprecationWarning: The tru
  if diff:
```



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 $sepalwidth \geq 1$. However, we also notice the non-linearity from the k-nearest neighbor classifier mixen 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 *GridSeach* object:

```
In [26]: mv_clf.get_params()
```

```
Out[26]: {'decisiontreeclassifier': DecisionTreeClassifier(class_weight=None, criterion='entropy
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort=False, random_state=0,
                      splitter='best'),
             'decisiontreeclassifier__class_weight': None,
             'decisiontreeclassifier__criterion': 'entropy',
             'decisiontreeclassifier__max_depth': 1,
             'decisiontreeclassifier__max_features': None,
             'decisiontreeclassifier__max_leaf_nodes': None,
             'decisiontreeclassifier__min_impurity_decrease': 0.0,
             'decisiontreeclassifier__min_impurity_split': None,
             'decisiontreeclassifier__min_samples_leaf': 1,
             'decisiontreeclassifier__min_samples_split': 2,
             'decisiontreeclassifier__min_weight_fraction_leaf': 0.0,
             'decisiontreeclassifier__presort': False,
             'decisiontreeclassifier__random_state': 0,
             'decisiontreeclassifier__splitter': 'best',
             'pipeline-1': Pipeline(memory=None,
                  steps=[('sc', StandardScaler(copy=True, with_mean=True, with_std=True)), ['clf',
                       intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                       penalty='l2', random_state=1, solver='liblinear', tol=0.0001,
                       verbose=0, warm_start=False)]]),
             'pipeline-1__clf': LogisticRegression(C=0.001, class_weight=None, dual=False, fit_inte
                       intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                       penalty='l2', random_state=1, solver='liblinear', tol=0.0001,
                       verbose=0, warm_start=False),
             'pipeline-1__clf__C': 0.001,
             'pipeline-1__clf__class_weight': None,
             'pipeline-1__clf__dual': False,
             'pipeline-1__clf__fit_intercept': True,
             'pipeline-1__clf__intercept_scaling': 1,
             'pipeline-1__clf__max_iter': 100,
             'pipeline-1__clf__multi_class': 'ovr',
```

```
'pipeline-1__clf__n_jobs': 1,
'pipeline-1__clf__penalty': 'l2',
'pipeline-1__clf__random_state': 1,
'pipeline-1__clf__solver': 'liblinear',
'pipeline-1__clf__tol': 0.0001,
'pipeline-1__clf__verbose': 0,
'pipeline-1__clf__warm_start': False,
'pipeline-1__memory': None,
'pipeline-1__sc': StandardScaler(copy=True, with_mean=True, with_std=True),
'pipeline-1__sc__copy': True,
'pipeline-1__sc__with_mean': True,
'pipeline-1__sc__with_std': True,
'pipeline-1__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', n_jobs=1,
            penalty='l2', random_state=1, solver='liblinear', tol=0.0001,
            verbose=0, warm_start=False)]],
'pipeline-2': Pipeline(memory=None,
     steps=[('sc', StandardScaler(copy=True, with_mean=True, with_std=True)), ['clf',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')]]),
'pipeline-2__clf': KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkow
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform'),
'pipeline-2__clf__algorithm': 'auto',
'pipeline-2__clf__leaf_size': 30,
'pipeline-2__clf__metric': 'minkowski',
'pipeline-2__clf__metric_params': None,
'pipeline-2__clf__n_jobs': 1,
'pipeline-2__clf__n_neighbors': 1,
'pipeline-2__clf__p': 2,
'pipeline-2__clf__weights': 'uniform',
'pipeline-2__memory': None,
'pipeline-2__sc': StandardScaler(copy=True, with_mean=True, with_std=True),
'pipeline-2__sc__copy': True,
'pipeline-2__sc__with_mean': True,
'pipeline-2__sc__with_std': True,
'pipeline-2__steps': [('sc',
  StandardScaler(copy=True, with_mean=True, with_std=True)),
 ['clf',
  KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
            metric_params=None, n_jobs=1, n_neighbors=1, p=2,
            weights='uniform')]]}
```

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:

```
In [27]: 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, scoring='roc_auc')
         grid.fit(X_train, y_train)
         for params, mean_score, scores in grid.grid_scores_:
             print('%0.3f +/- 0.2%f %r' %
                   (mean_score, scores.std() / 2, params))
         print('Best parameters: %s' % grid.best_params_)
         print('Accuracy: %0.2f' % grid.best_score_)
```

```
0.933 +/- 0.20.066898 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 0.001}
0.947 +/- 0.20.066667 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 0.1}
0.973 +/- 0.20.033333 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 100.0}
0.947 +/- 0.20.066667 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C': 0.001}
0.947 +/- 0.20.066667 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C': 0.1}
0.973 +/- 0.20.033333 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C': 100.0}
Best parameters: {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 100.0}
Accuracy: 0.97


/usr/lib/python3.6/site-packages/sklearn/model_selection/_search.py:761: DeprecationWarning: The
  DeprecationWarning)
```

As we can see, we get the best cross-validation results when we choose a lower regularization strength (*C=100.0*), 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**.

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-layer ensemble, where the first layer consists of individual classifiers that feed their predictions to the second level, where another classifier (typically logistic regression) is fit to the level 1 classifier predictions to make the final predictions.

## 3 Bagging - building an ensemble of classifiers from bootstrap samples

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

### 3.1 Bagging in a nutshell

To provide a more concrete example of how the bootstrapping 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:

As we can see from the previous illustration, each classifier receives a random subset of samples from the training set. Each subset contains a certain portion of duplicates and some of the original samples do not appear in a resampled dataset at all due to the 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. In fact, random forests are a special case of bagging where we also use random feature subsets when fitting the individual decision trees.

### 3.2 Applying bagging to classify samples of the Wine dataset

To see bagging in action, let's create a more complex classification problem using the Wine dataset. Here, we will only consider the Wine classes 2 and 3, and we select two features: *Alcohol* and *OD280/OD315 of diluted wines*:

```
In [28]: 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 = pd.read_csv('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 encode the class labels into binary format and split the dataset into 80 percent training and 20 percent test sets, respectively:

```
In [29]: 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)
```

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:

```
In [30]: from sklearn.ensemble import BaggingClassifier
         from sklearn.tree import DecisionTreeClassifier

         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 dataset to compare the perfomance of the bagging classifier to the performance of a single unpruned decision tree:

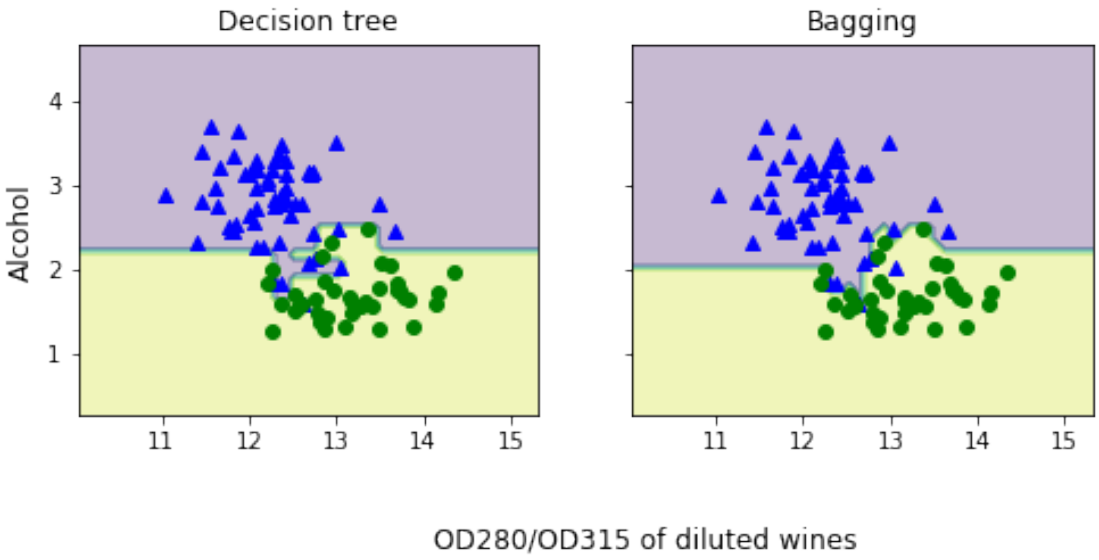
```
In [31]: from sklearn.metrics import accuracy_score

         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 predict all the class labels of the training samples correctly; however, the substantially lower test accuracy indicates high variance (overfitting) of the model:

```
In [32]: 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 set (both 100 percent), we can see that the bagging classifier has a slightly better generalization performance, as estimated on the test set. Next, let's compare the decision regions between the decision tree and the bagging classifier:

```
In [33]: import matplotlib.pyplot as plt
         import numpy as np

         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.text(10.2, -1.2, s='OD280/OD315 of diluted wines',
                  ha='center', va='center', fontsize=12)
         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 dataset's high dimensionality can easily lead to overfitting in single decision trees, and this is where the bagging algorithm can really play to it strengths. Finally, we shall note that the bagging algorithm can be an effective approach to reduce the variance of the model. However, bagging is ineffective in reducing model bias, that is, models that are too simple to capture the trent 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.

# 4 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, **AdaBoost (Adaptive Boosting)**.

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 of random guessing - a typical example of a weak learning is a decision tree stump. The key concept behind boosting is to focus on training samples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training samples to improve the performance of the ensemble.

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

## 4.1 How boosting works

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

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

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.

In contrast to the original boosting procedure as described here, AdaBoost uses the complete training set to train the weak learners where the training samples 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 start with subfigure 1, which represents a training set for binary classification where all training samples are assigned equal weights. Based in 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 circle), as well as possibly 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 - 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 assinged 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.

## 4.2 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:

```
In [34]: 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 we can see, the decision tree seems to underfit the training data in contrast to the unpruned decision tree that we saw in the previous section:

```
In [35]: 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
```

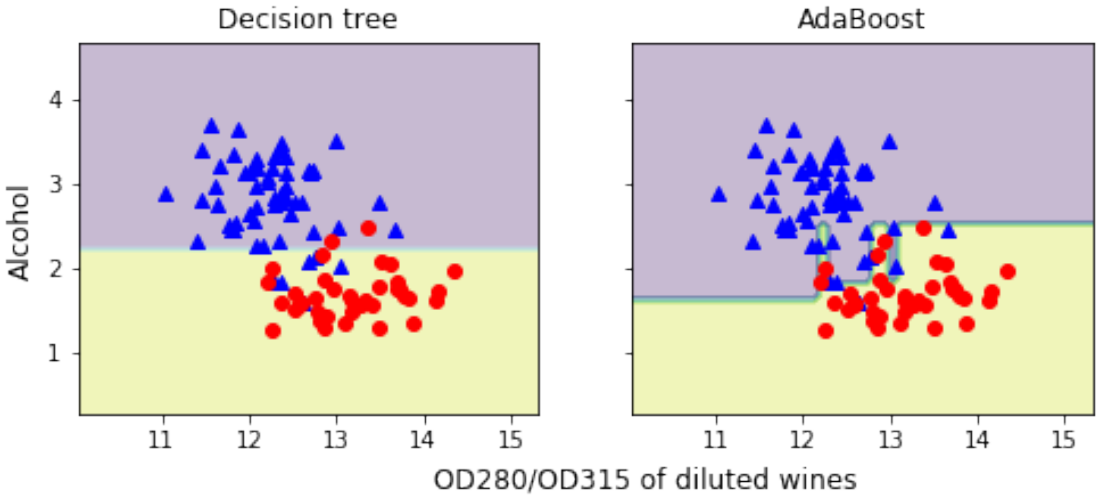
As we can see, the AdaBoost model predicts all class labels of the training set correctly and also shows a slightly improved test set performance compared to the decision tree stump. However, we also see that we introduced additional variance by our attempt to reduce the model bias - a higher gap between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

Although we used another example for demonstration purposes, we can see that the performance of the AdaBoost classifier is slightly improved compared to the decision tree stump and achieved the very similar accuracy scores as the bagging classifier that we trained in the previous section. However, we shall note that it is considered bad practice to select a model based on the repeated usage of the test set. The estimate of the generalization performance may be over-optimistic.

Lastly, let us check what the decision regions look like:

```
In [36]: 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, ada],
                                 ['Decision tree', 'AdaBoost']):
             clf.fit(X_train, y_train)
```

```
        Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        axarr[idx].contourf(xx, yy, Z, alpha=0.3)
        axarr[idx].scatter(X_train[y_train==0, 0],
                           X_train[y_train==0, 1],
                           c='blue', marker='^')
        axarr[idx].scatter(X_train[y_train==1, 0],
                           X_train[y_train==1, 1],
                           c='red', marker='o')
        axarr[idx].set_title(tt)
    axarr[0].set_ylabel('Alcohol', fontsize=12)
    plt.text(10.2, -0.5, s='OD280/OD315 of diluted wines',
             ha='center', va='center', fontsize=12)
    plt.show()
```




By looking at the decision regions, we can see that the decision boundary of the AdaBoost is substantially more complex than the decision boundary of the decision stump. In addiction, we note that the AdaBoost model separates the feature space very similar 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 computation complexity compared to individual classifiers. In practice, we need to think carefully about whether we want to pay the price of increased computation costs for an often relatively modest improvement in predictive performance.

## 5 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 to reduce the variance of a model by drawing random bootstraps samples from the training set 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.